Worst-case temporal analysis of real-time dynamic streaming applications

Worst-case Temporal Analysis of Real-time Dynamic Streaming Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 19 november 2014 om 16.00 uur

door

Firew Merete Siyoum

geboren te Addis Ababa, Ethiopië

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.C.P.M. Backx
1e promotor:	prof.dr. H. Corporaal
copromotor:	dr.ir. M.C.W. Geilen
leden:	prof.dr.ir. M.J.G. Bekooij (University of Twente, NXP)
	prof.dr.ir. C.H. van Berkel
	prof.dr. A. Jantsch (Royal Institute of Technology)
	dr. A.D. Pimentel (University of Amsterdam)

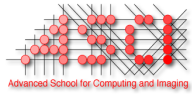
Worst-case Temporal Analysis of Real-time Dynamic Streaming Applications

Firew Merete Siyoum

Worst-case Temporal Analysis of Real-time Dynamic Streaming Applications
By Firew Merete Siyoum, 2014
A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3716-7

Committee:

prof.dr.ir. A.C.P.M. Backx (*chairman*, Eindhoven University of Technology)
prof.dr. H. Corporaal (*promotor*, Eindhoven University of Technology)
dr.ir. M.C.W. Geilen (*copromotor*, Eindhoven University of Technology)
prof.dr.ir. M.J.G. Bekooij (University of Twente, NXP)
prof.dr.ir. C.H. van Berkel (Eindhoven University of Technology, Ericsson)
prof.dr. A. Jantsch (Royal Institute of Technology (KTH))
dr. A.D. Pimentel (University of Amsterdam)



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 313.

The work presented in this thesis has been partially supported by SenterNovem (an agency of the Dutch Ministry of Economical Affairs), as part of the EUREKA/CATRENE/COBRA project under contract CA104.

Printing: Ipskamp Drukkers B.V., Enschede, The Netherlands

Abstract

Contemporary embedded wireless and multimedia applications are typically implemented on a Multiprocessor System-on-Chip (MPSoC) for power and performance reasons. The MPSoC commonly comprises heterogeneous resources that are shared between multiple applications under different scheduling policies. These applications have strict real-time constraints such as worst-case throughput and maximum end-to-end latency. It is crucial to guarantee that such constraints are satisfied at all operating conditions. Simulation and measurement-based analysis techniques cannot guarantee worst-case temporal bounds, since it is impractical to cover all possible system behaviors. Thus, analytical techniques are often used to compute conservative temporal bounds. In particular, dataflow models of computation (MoCs) have been widely used to model and analyse streaming applications.

A challenge to dataflow-based design-time analysis of present-day streaming applications is their dynamic execution behavior. These applications change their graph structure, data rates and computation loads, depending on their operating modes. A conservative static dataflow model, such as Synchronous Dataflow (SDF), abstracts from such varying operating modes for the sake of analysability. However, the abstraction leads to overly pessimistic temporal bounds. This further leads to unnecessarily large resource allocations to guarantee real-time latency and throughput requirements. Thus, a refined temporal analysis that considers the different operating modes is crucial to compute tight real-time temporal bounds and, consequently, avoid unnecessary overallocation of scarce MPSoC resources. Moreover, the temporal analysis should be fast enough to efficiently explore the application mapping design-space through an iterative process. To that end, this thesis presents a number of contributions that form a framework to analytically determine real-time temporal bounds of streaming applications that are mapped onto a heterogeneous MPSoC platform.

The analysis framework uses the *Scenario-aware Dataflow (SADF)* MoC, which explicitly models each static operating mode, called *scenario*, of a dynamic application with a SDF graph. Furthermore, it captures all possible sequences of scenario executions by the language of a finite-state machine (FSM).

The thesis begins with an in-depth study of intra-application dynamism in modern-day streaming applications. The investigation conducts case studies on different applications, such as LTE, which is a recent cellular connectivity standard, and MPEG4 video decoder. The case studies demonstrate the benefits of capturing intra-application dynamism through SADF for tighter temporal analysis. The case studies also reveal that identification of all scenarios and scenario sequences can be challenging because of the large number of possible scenarios. This thesis addresses this challenge with an automated approach that extracts a scenario-based analysis model for a class of parallel implementations, called Disciplined Dataflow Network (DDN). The extraction process identifies all possible scenarios of a DDN and employs state-space enumeration to determine all possible sequences of executions of these scenarios. The result is an FSM-based SADF analysis model. The approach is demonstrated for the CAL actor language and has been implemented in an openly available CAL compiler.

Once a SADF model is constructed, it is mapped onto a heterogeneous MPSoC platform and resources are allocated, while satisfying real-time constraints such as throughput and end-to-end latency. The thesis makes the following major contributions in this respect. First, it generalizes the existing throughput analysis technique of SADF to support *self-timed unbounded* scenarios as well as arbitrary inter-scenario synchronizations through *data-dependency actors* and *initial token labeling*. The generalization lifts existing restrictive assumptions such as self-timed boundedness and synchronizations limited to initial tokens on identical channels of scenarios. A byproduct of the generalized throughput analysis technique is an approach to verify boundedness of FSM-based SADF models.

Another contribution is a faster and tighter approach to analyse application mappings. The new approach, called *Symbolic Analysis of Application Mappings*, avoids constructing resource-aware dataflow models, which are often used in existing approaches. The new technique combines symbolic simulation in Max-plus algebra with worst-case resource curves. As a result, it keeps the graph size intact and improves scalability, which makes it tens of times faster than the state-of-the-art. Moreover, it gives tighter temporal bounds by improving the worst-case response times of requests that arrive in the same busy time of a resource.

The final major contribution is an approach to derive the maximum end-to-end latency of applications mapped onto a shared platform. The approach derives a bound to the maximum end-to-end latency under a periodic source and sketches how to address aperiodic sources, such as sporadic and bursty input streams.

The contributions form an analysis framework that takes a high-level DDN specification of a dataflow application as an input and then 1) automatically constructs an FSM-based SADF dataflow model, 2) verifies basic properties such as deadlock-freedom and boundedness, and 3) derives real-time temporal bounds such as worst-case throughput and end-to-end latency, while considering resource sharing in a heterogeneous MPSoC platform. This thesis illustrates this flow with case-study applications. The contributions advance the state-of-the-art in terms of accuracy, scalability, model expressiveness as well as ease of use.

Contents

CHAPTER 1

Introduction

Advancements in computer technologies are continuously changing the life-styles of our modern society. Computers are now tightly linked to most of our daily lives. We use general-purpose desktop and laptop computers on a daily basis at home and at work for communication, entertainment, Internet browsing and office productivity. The majority of computers are, however, *embedded systems* that are integrated into many devices to carry out dedicated functionalities. Embedded systems are now core entities in consumer electronics, automotive, avionics, home appliances, medical appliances and so many others. Parallel to their vast applicability, the complexity of embedded systems also varies widely from a light-weight microcontroller in a sensor node to heavy-weight multiprocessor systems running full-fledged operating systems, like Android and Windows. Nonetheless, the pervasive presence of embedded systems is felt nowhere stronger than connectivity and multimedia domains. Connectivity is all around us more than ever, with an expected 4.5 billion mobile phone users and 1.7 billion smart-phone users world-wide in 2014 [24]. Multimedia-rich communication, information and entertainment are at our fingertips through high-tech consumer electronics such as smart-phones, high-definition television sets, gaming consoles, digital cameras, MP3 players and CD/DVD/Blue-ray players. Embedded multimedia and wireless systems lie at the heart of these devices. These types of embedded applications are often referred to as *embedded streaming applications*, as they are characterized by a continuous processing of data streams such as packets and frames.

The central focus of this thesis is the design of embedded streaming systems. This chapter introduces major design challenges and outlines the key contributions made by this thesis to address them. The chapter is organized in five sections. Section 1.1 discusses current trends and main design challenges of embedded streaming applications. Section 1.2 outlines the approach taken by this thesis to tackle these design challenges. Section 1.3 lists the key contributions of the thesis. Section 1.4 presents the organization of the rest of the thesis.

1.1 Embedded Streaming Applications

In the early days, mobile phones were designed only for voice communication. Today, high-feature cellular phones integrate much more functionalities such as media playing, gaming, browsing, navigation, messaging, digital imaging and others. Many of these functionalities have multimedia content and require wireless connectivity. As a result, high-feature phones support a wide-range of multimedia codec and wireless communication standards. Figure 1.1 shows an example of a board-level view of a high-feature cellular phone. The figure shows that such systems include multimedia processors for video and audio capturing, recording and playback. They also have a number of baseband processing blocks for wireless connectivity, such as WiFi, Bluetooth and 3G/4G cellular modems. Multimedia processing involves the coding and decoding of digital audio and video streams. Popular examples are the different MPEG-x standards from the Moving Picture Experts Group (MPEG) and H.26x standards from the Video Coding Experts Group (VCEG). Wireless communication standards are also required for different purposes such as 3G/4G cellular connectivity (e.g. WCDMA, HSDPA, LTE), wireless connectivity (e.g. IEEE 802.11a/b/g/n), digital video and audio broadcasting (e.g. DVB, DAB) and GPS navigation. The above mentioned embedded multimedia and wireless applications are also available in many other consumer electronics such as DVD, Blue-ray and MP3 players, video cameras, set-top boxes, television sets, automotive entertainment units and navigation systems.

A characterizing feature of embedded multimedia and wireless applications is that they process a continuous stream of input data and produce a stream of output data. As a result, they are often referred to as *embedded streaming applications*. A data stream can be a stream of packets/frames in wireless applications, or a sequence of image frames or a compressed bitstream in video codecs. Data processing often involves multiple pipelined signal processing stages, where the output of one stage is fed to the next. Hence, data processing is primarily data-driven, i.e. the different stages are activated by the arrival of data.

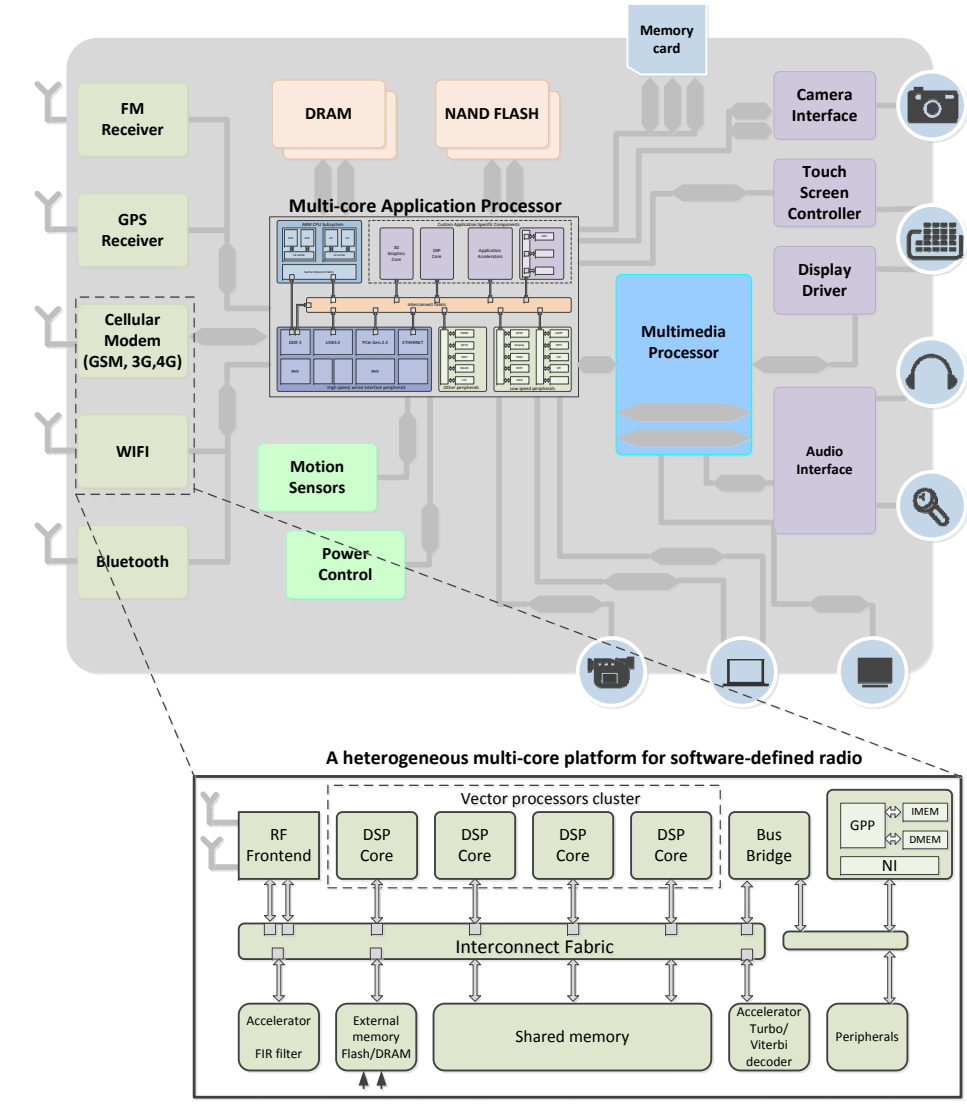


Figure 1.1: An example of a board-level view of a high-feature cellular phone (based on [61]). At the center of the system lies a multi-core application processor that includes a general-purpose multiprocessor, a graphics processor, hardware accelerators and peripherals. The system also includes a number of baseband processors for wireless connectivity. Baseband processor architectures (cf. Section 1.1.3) also combine homogeneous and heterogeneous multiprocessing, as shown by the figure at the bottom.

1.1.1 Trends in Streaming Applications

The current trend of embedded streaming systems shows that multiple applications are being integrated into the same device. These applications are started and stopped at run-time. Adaptivity to different quality requirements and resource availability (e.g. bandwidth and power) is also highly demanded. These trends are better seen in two emerging technologies from embedded wireless and multimedia domains: *software-defined radio (SDR)* and *reconfigurable video coding (RVC)*, as discussed in the following two sections.

Software-defined Radio

Convergence of application domains and differences in technical merits of standards are demanding contemporary radio receivers to support various radio standards and run multiple applications simultaneously. For instance, smart phones need to support various cellular communication standards (such as GSM, WCDMA and LTE), broadcast radio and television standards (such as DAB and DVB) and wireless connectivity standards (such as WLAN 802.11x and Bluetooth). Traditionally, the physical layer functionality of radios is implemented with hardware blocks. The main advantage of such hardware-based radio designs is performance and low power consumption. However, they have very low flexibility to catch up with the continuously evolving and growing technological advancements at low cost. A dedicated hardware baseband block per standard also impose high design and production cost.

As opposed to fully hardware-based solutions, Software Defined Radio (SDR) is a radio where some or all of the physical layer functionalities are implemented as software processes that run on a Multiprocessor System-on-Chip (MPSoC) architecture platform. Software Defined Radio (SDR) brings flexibility and, ultimately, cost efficiency in the design of these multi-functional wireless communication devices. SDR allows manufacturers to introduce new multi-band and/or multi-functional wireless products into the market at low design cost. It also reduces maintenance and support cost as software upgrades, new features and bug-fixes can be easily provided to existing radio systems.

An important aspect of SDR design for baseband processing is the ability to support multiple simultaneously running applications. These applications may also be started or stopped at any time. This results in different use-cases, which may result in a dynamically changing workload [12, 59]. Furthermore, intra-application dynamism, from within a single radio, also cause significant workload variations. Such dynamism may come from a radio's adaptation to resource availability. For instance, 3GPP's Long Term Evolution (LTE), which is a pre-4G cellular standard, uses adaptive modulation and coding (AMC) to dynamically

adjust modulation schemes and transport block sizes to adapt to varying channel conditions. Intra-application dynamism may also come from the different modes of operation of data processing. For instance, according to the discussion in [59], WLAN packet decoding consists of four different modes, namely *Synchronization, Header decoding, Payload decoding* and *Cyclic-redundancy check*. Once a packet is detected, *Synchronization* mode is executed repeatedly until it succeeds. Then, *Header decoding* decodes the packet header to determine the size of the payload that may vary from 1 to 256 OFDM symbols. After header decoding, *payload decoding* is executed as many times as the number of OFDM symbols. Finally, *cyclic redundancy check* is performed and an acknowledgment packet is sent. These modes may activate different sets of tasks that may lead to variations in the computational workload.

Reconfigurable Video Coding

The different MPEG video coding standards have enjoyed huge acceptance since their inception in 1988. Over the years, the standards are becoming richer in syntax and tools, targeting higher quality and compression ratio. This is in turn making the standards increasingly complex and time-taking to produce. In the past, standards were specified through a monolithic textual specification and a sequential C/C++ reference implementation [14]. This kind of monolithic specification hinders reusability by making use of the significant overlap between successive standards. This means, adding new coding tools to a standard requires a new specification for which all components are modified, even though only a few tools and interfaces are changed. Another drawback of a monolithic specification is that it does not consider the effort needed for a parallel implementation on multi-core hardware platforms. As a result, video devices typically support a single profile or a few selected profiles of a specific standard. Consequently, they have limited adaptivity to different application needs, quality requirements and resource availability.

These observations led to the development of the Reconfigurable Video Coding (RVC) standard [58]. RVC aims at providing a model of specifying MPEG standards at higher-level than the one provided by monolithic C based specifications. At the core of the RVC standard are the CAL dataflow language [23] and a library of video coding tools. MPEG standards are then specified by constructing a network of standard components taken from the library. The resulting specification is compact, modular and exposes the intrinsic concurrency of the video coding application. The modularity facilitates the design of reconfigurable video codecs by replacing and reconnecting components at run-time in a plug and play manner. RVC further provides new tools and methodologies for describing bitstream syn-

taxes of dynamically configurable codecs. Exposing intrinsic concurrency through a dataflow language paves the way to an efficient parallel implementation on a multi-core hardware platform. The parallel specification is a better starting point than sequential C/C++ reference software, as it opens the opportunity for rapid parallel implementations through automatic code generation tools such as CAL2C and CAL2HDL.

In summary, current trends in streaming applications show that there is a demand to support multiple standards or functionalities on the same device. Applications are started and stopped at run-time. Reconfigurability and adaptivity are also required to satisfy different application quality requirements and adapt to resource availability and environmental conditions.

1.1.2 Real-time Properties

Functionally correct processing (coding and decoding) of input data streams is not sufficient for a correct implementation of an embedded streaming application. It is also crucial *when* the processing is completed. This is because these applications have timing requirements that determine their proper functionality. A video decoder has to diligently feed the display a preset number of frames per second to meet the desired quality requirement. In wireless applications, the rate at which packets must be processed is dictated by standards. Furthermore, wireless standards have strict maximum timing requirements to acknowledge (respond to) a properly received packet. Due to such strict requirements of timely operations, embedded streaming applications are categorized as *real-time* applications.

Two key real-time temporal requirements of embedded streaming applications are *throughput* and *latency*. Throughput defines the rate at which data is processed, such as the number of video frames or wireless packets per time-unit. These are mostly dictated by standards. For instance, LTE has a frame structure, which has 10 sub-frames and is $10msec$ long. This gives an LTE receiver a throughput requirement to handle processing of at least one sub-frame every millisecond. Modern video cameras are also required to support standard video frame rates such as 24, 25 or 30 frames per second. Latency defines the maximum end-to-end duration between the arrival of an input data and the completion of its processing. In WLAN 802.11a, for instance, an acknowledgment packet must be sent within $16\mu sec$ of a successful packet reception. This time guard of $16\mu sec$, known as the Short Intra-Frame Spacing, is a latency constraint that must be satisfied. Latency is a requirement that must be met by every individual sample. Throughput, on the other hand, typically deals with the long-run average rate, irrespective of arrival or production jitter, which can be smoothed through buffering and selecting an appropriate sampling rate.

The real-time embedded community categorizes real-time applications as *soft* or *hard*, depending on the severity of the consequence of failure to met timing constraints. Hard real-time applications are often defined as critical systems, where timing deadlines must be met at all times, such as vehicle airbag system, artificial cardiac pacemaker or industrial process control. In soft real-time applications, a limited set of deadline misses are tolerated, at a price of degraded quality of service, such as artifacts in a decoded video and clicks in audio playbacks. A third categorization, called *firm* real-time, is also sometimes used, in which infrequent deadline misses are tolerated but a result becomes useless after its deadline. In spite of such classifications, we believe that the class of a real-time application is a designer's choice when it comes to a specific implementation. For instance, in audio codecs, intermittent clicks due to sample dropping can be totally unacceptable in today's competitive market. SDR devices cannot be certified unless they meet all timing requirements and be compliant with their respective standards.

1.1.3 Heterogeneous Multi-core Architectures

The computational workload of embedded streaming applications is ever increasing, along with their rising quality of service, such as higher resolutions and data rates. For instance, LTE specifies a downlink rate of at least 300 Mbit/s and an uplink of at least 75 Mbit/s. It is a pre-4G standard, a step towards its successor, LTE-Advanced (LTE-A) whose specifications are expected to require a peak data rate of 1 Gbit/s and higher quality of service [18]. Moreover, hand-held embedded devices are battery-operated and, as a result, are power-constrained. Consequently, multi-core hardware architectures have become the ultimate designers' option to support the high-performance and low-power requirements of these systems. Other key drivers for multi-core architectures are the much needed flexibility and reconfigurability, in areas like SDR and RVC.

Multi-core architectures for embedded streaming applications combine homogeneous and heterogeneous multiprocessing. They employ devices including general purpose processors (GPP), digital signal processors (DSP) and application-specific programmable accelerators. An example architecture for SDR is shown by the bottom figure of Figure 1.1, which is intended to run different wireless standards. An overview of existing architectures and demonstrators for SDR can be obtained in [41], which also shows the above discussed trends.

The heterogeneity enables to achieve lower-power and high-performance architectures using specialized cores, while offering flexibility in a balanced manner. General-purpose cores (e.g. ARM) are used for handling protocols and control tasks. A set of DSP cores (e.g. EVP [11]) are used for signal and data processing algorithms, such as synchronization, channel estimation and demodulation, where

flexibility is valuable. A set of weakly-programmable hardware accelerators are used when flexibility is of limited value [10]. For instance, a Multi-Standard Multi-Channel decoder is a weakly-programmable core that consists multiple reconfigurable Hardware Units (HUs). The core allows limited programmability as HUs can be reconfigured at run-time to handle different radio standards [93].

The move to heterogeneous multi-core architectures addresses the power and performance issues by creating multiple specialized processing cores that execute at lower clock frequencies. Nevertheless, the desired high-performance and low-power design can not be realized without effectively exploiting the parallelism offered by such platforms. This means the design challenge heavily shifts to the software domain as well as to the high-level system dimensioning and scheduling phases, as further discussed next in Section 1.1.4.

1.1.4 Design Challenges

As highlighted in Section 1.1.1 and 1.1.3, contemporary embedded streaming systems are required to support multiple applications. The computational workload of applications is also increasing due to higher quality of service requirements. As a result, these systems are becoming increasingly complex to design. Time-to-market is also shortening due to strong competitions, as market opportunities will be missed if a product is delayed. The industry's response to cope with these challenges is the *platform-based design*. Platform-based design tackles the design complexity by reusing pre-designed *Intellectual Property (IP)* components to develop a platform that is suitable for a certain application domain. This shortens the time-to-market and leads to highly advanced system designs, as it allows vendors to focus on their core competence, while integrating refined and matured IPs from other vendors into their products. It also helps to reduce non-recurring engineering costs, since the development of new IPs requires significant investment due to high costs of designers, tools, infrastructures and mask making.

Platform-based design of embedded systems commonly follows the Y-chart approach, as shown in Figure 1.2. The design begins with a given set of applications and a multi-core architecture platform template. The design goal is to instantiate a platform and dimension resources such that design and performance requirements of all applications are met. This requires mapping applications onto a platform instance. The mapping involves scheduling tasks and allocating computation, communication and storage resources. The mapping is followed by analysing and evaluating the mapping decisions to verify if requirements are met [64]. This is in general an intensive *design-space exploration (DSE)* process. It requires repeated revisions of application specifications and platform instantiations, until requirements are satisfactorily met with minimized cost.

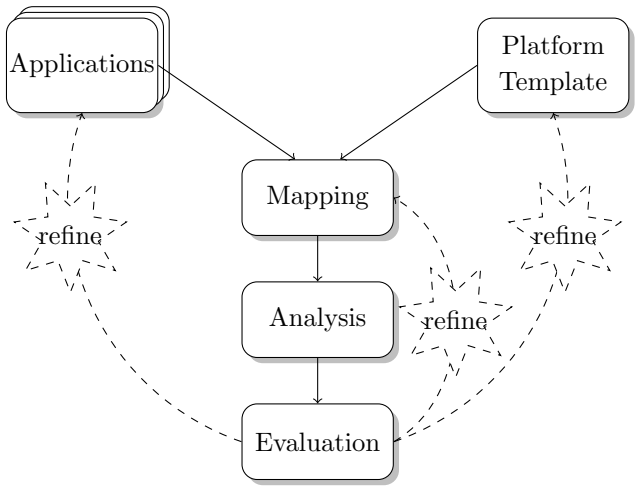


Figure 1.2: The Y-chart approach

The platform-based design of contemporary embedded streaming applications incurs a number of challenges. The first one is *challenge to predictable design*. These applications have real-time temporal requirements, such as latency and throughput, as discussed in Section 1.1.2. These temporal requirements come from standard specifications. To comply with standards, the design of such systems must guarantee that temporal requirements are met, even at worst-case conditions. Ensuring predictability is challenged by the complexity and dynamism of applications that lead to data-dependent resource requirements (cf. Section 1.1.1). In LTE, for instance, physical layer resource allocations of data and control channels dynamically change across frames, depending on varying channel conditions (cf. Chapter 4). Designs that do not consider dynamism may have to rely on static worst-case assumptions that give pessimistic results. In such cases, MPSoC resources, such as processors and memories, have to be *over-allocated* to ensure predictability. Over-allocation of resources brings us to the second design challenge: *challenge to resource-constrained design*. MPSoC resources are scarce and must be efficiently utilized to accommodate the increasingly high workload of latest streaming applications. E.g., the digital workload of high-feature cellular phones tops $100GOPS$ that must be accommodated under a power budget of $1Watt$ [10]. This requires aggressive resource allocation and mapping strategies.

Consequently, the design of modern-day embedded streaming applications requires a systematic approach that (1) abstracts the system complexity, (2) allows temporal analyzability to guarantee strict real-time requirements, and (3) is able to capture the dynamism of applications to avoid over-allocation of resources.

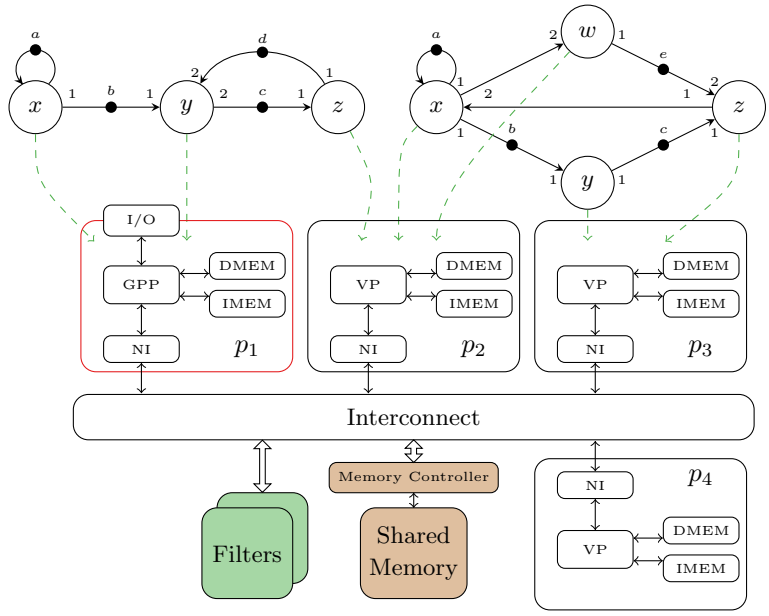


Figure 1.3: A heterogeneous MPSoC shared by multiple applications. The figure shows two application models mapped onto the same MPSoC platform.

1.2 Our approach

This thesis makes contributions towards addressing the design challenges of embedded streaming systems through a *predictable system design* methodology. A predictable system is defined as a system whose timing behavior can be reasonably bounded [7, 9, 56, 85]. A predictable system design aims at guaranteeing at design-time that an application will meet its timing constraints. It also targets verifying basic properties such as deadlock-freedom and memory boundedness. A predictable system design requires architectures, application specifications, schedulers and techniques, which allow analysing timing behavior. Following the platform-based design paradigm, we assume a given set of applications are intended to be mapped on a heterogeneous MPSoC platform, illustrated in Figure 1.3. The platform comprises a set of *processor tiles*, which may have local instruction and data memories (DMEM and IMEM), but have no caches as they impede reasonable timing bounds. Processor tiles are connected through a predictable interconnect [38] that offers each connection a guaranteed bandwidth and maximum latency. Processor tiles may include general-purpose cores (GPP), vector processors (VP) and dedicated accelerators (e.g. filters).

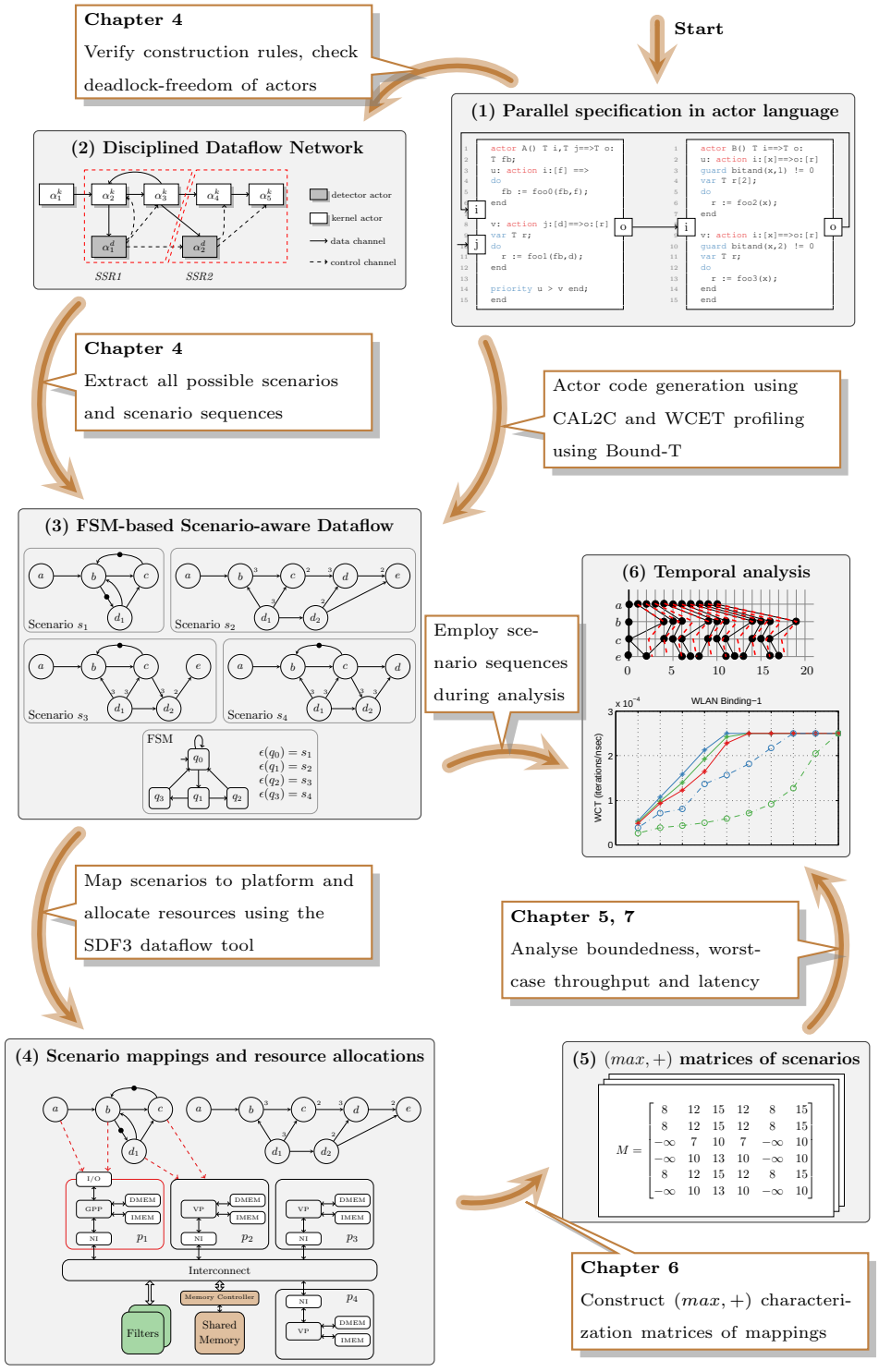


Figure 1.4: Analysis framework for dynamic embedded streaming applications

The predictable system design instantiates a platform and allocates resources to applications such that all real-time requirements are guaranteed to be satisfied. Figure 1.4 shows the design framework proposed in this thesis. The framework aims at a predictable design of embedded systems, which comprise multiple dynamic streaming applications that are mapped onto a shared MPSoC. The starting point is a parallel specification of an application. The specification is a network of dynamic tasks, which may change their input and output data rates between executions. The goal is then to verify basic properties, such as deadlock-freedom and boundedness, and real-time constraints, such as throughput and end-to-end latency. The presented analysis framework achieves this through a *model-driven design* strategy that allows *formal temporal analysis* and *automation*. These three aspects are further discussed next in Section 1.2.1, 1.2.2 and 1.2.3, respectively.

1.2.1 Model-driven Design

Streaming applications process a continuous stream of input data. It is essential to guarantee that these applications can execute *ad infinitum* without a deadlock. Moreover, they need to operate also in a bounded buffer space, a property known as *boundedness*. Thus, it is crucial to guarantee basic properties such as deadlock-freedom and boundedness. This is challenging, since modern embedded streaming applications consist of complex parallel programs with significant dynamism. An effective strategy is to abstract from implementation details through high-level *analysis models* [47]. Such models selectively capture important aspects that are required for design-time verification of basic properties. Dataflow models of Computation (MoCs) have been shown to be effective in this regard to model streaming applications at a higher level of abstraction [96, 101, 105]. A dataflow MoC consists of a set of *actors*, which encapsulate computational units. Actors communicate by sending data *tokens* through their *ports* in a message-passing manner through First-In-First-Out (FIFO) buffer *channels*. Such a representation is in-line with the data-driven execution of these applications. Dataflow MoCs are effective means in verifying basic properties, as they abstract from unnecessary implementation details, while exposing concurrency and synchronization aspects. This may enable efficient parallel implementations on MPSoC platforms.

Today, there exists various types of dataflow MoCs which vary with their level of expressiveness and analyzablity [85]. Synchronous Dataflow (SDF) [53], for example, has gained broad acceptance in design tools due to its analyzablity. Figure 1.3 shows two SDF graphs that are mapped on a MPSoC. The black-dots in the figure are initial tokens of channels. The numbers on the edges indicate data rates of ports. A SDF actor *fires*, i.e. starts execution, by consuming from each of its input ports as many tokens as the port rate. At the end of the execution,

which takes a given (worst-case) amount of time, it produces at each of its output ports as many tokens as the port rate. The throughput of a *self-timed bounded* SDF graph is analyzed by state-space exploration [35] or $(max, +)$ spectral analysis [28]. A SDF graph is self-timed bounded if the number of tokens in every channel is bounded in a self-timed execution. Self-timed execution is of special interest as it gives the maximum achievable throughput of a SDF graph [34]. Necessary and sufficient conditions for deadlock-free execution of SDF as well as schedulability with bounded buffer space are studied in [34, 52, 53]. SDF actors consume and produce fixed number of tokens per execution. As a result, SDF is too static[1] to capture the dynamic behavior of modern-day multimedia and wireless applications. A static SDF model of a dynamic application has to capture the worst-case behavior across all modes. However, such abstraction may lead to overly pessimistic temporal bounds. This further leads to unnecessarily large budget reservation of resources such as processors and communication interconnects to guarantee real-time latency and throughput requirements. Thus, a refined temporal analysis that considers the different operating modes of an application is crucial to compute tight temporal bounds and, consequently, avoid unnecessary over-allocation of scarce MPSoC resources.

Different dataflow models are proposed that enhance the expressiveness of SDF [37, 48, 59, 66, 89, 95, 97, 104]. The majority of them, however, are either not sufficiently analysable or do not have known design-time temporal analysis techniques at all (cf. Section 3.4 for more). Our analysis framework uses the FSM-based Scenario-aware dataflow (FSM-SADF) MoC to model dynamic streaming applications. FSM-SADF is introduced in [28] to improve the expressiveness of SDF, while allowing for design-time analysability. FSM-SADF splits the dynamic data processing behavior of an application into a group of static modes of operation. Each static mode of operation, referred to as *scenario*, is modeled by a SDF graph. An FSM-SADF may dynamically change scenarios. The possible orders of executions of these scenarios are specified by a finite state machine (FSM).

FSM-SADF is expressive enough to capture dynamism in streaming applications. It allows scenarios to have different graph structures as well as varying port rates and actor execution times. It also enables a more accurate design-time analysis of dynamic streaming applications, capitalizing on the analysis techniques of static SDF. It exploits the sequence of scenario executions encoded by the FSM to avoid unnecessarily pessimistic analyses. For instance, if scenario sequences are not considered, consistency and boundedness of the application can only be guaranteed if every scenario of the application is also consistent and bounded. This condition is unnecessarily constraining. With scenario sequences, it is sufficient

[1] Chapter 3 discusses this challenge in further detail with case-study applications.

to show that all scenario sequences within cycles of the FSM are bounded and consistent, even if the individual scenarios are not [31, 79].

1.2.2 Automation

Properties guaranteed on a dataflow model are only useful as long as the implementation remains consistent with the model of the system. Otherwise, the derived guarantees apply only to the model and serve no purpose! Constructing an FSM-SADF analysis model and maintaining its consistency throughout the design cycle is not a trivial process. First, the analysis model abstracts from implementation details such as how scenario switching is decided. This means some important implementation aspects, such as scenario detection, have to be addressed, to define the types of parallel implementations for which such a model can be constructed. Second, the validity of abstraction of the analysis model must be verifiable. Third, modern-day streaming applications have a large number of possible scenarios, which makes manual model construction unattractive. It is time-consuming, error-prone and requires constant revisions to maintain consistency with changes of the application.

This thesis addresses this challenge with an automated approach that extracts a scenario-based analysis model. The input to the extraction process is a parallel implementation of the application, written in a concurrent language, illustrated at (1) in Figure 1.4. The extraction technique is largely language-independent, since it employs Dataflow Process Networks (DPN) [54] to characterize a parallel implementation of an application. DPN has been introduced to give a common denotational semantics to concurrent languages. A DPN is a network of actors that communicate by message-passing through FIFO buffers. Each actor has a set of different *firings*. Each firing consumes and produces a fixed number of data *tokens*. Executions of the firings are controlled by *firing rules* that specify the conditions for the execution of these firings. These conditions may be *data-dependent* and *state-dependent*, i.e. they may depend on values of input tokens and actor state. Thus, a DPN actor may have data-dependent token production and consumption rates.

DPNs are expressively Turing-complete, and hence, it is not always possible to construct a scenario-based analysis model for arbitrary DPNs. We introduce a class of parallel implementations, which we call *Disciplined Dataflow Network (DDN)* (illustrated at (2) in Figure 1.4), for which construction of a scenario-based model is guaranteed to be possible. Moreover, a construction process is defined and automated. The goal of DDN is to define construction rules that enforce a well-defined structure on the *control flow* that determines scenarios of a parallel implementation. To that end, DDN differentiates between *detector* and *kernel* ac-

tors [89]. Detectors are the initiators of variations in dynamic network behaviors, while kernels are the followers. To keep models analysable, DDN restricts data and state dependencies of actors. For instance, it restricts the state-dependency of kernels to a finite set of states and their data-dependencies to control tokens from detectors. Compliance of an input program with such construction rules can be automatically checked.

The automated extraction framework identifies all possible scenarios of a DDN and extracts their SDF graphs. It then derives all possible sequences of executions of these scenarios through state-space enumeration and constructs a finite-state machine to characterize the scenario sequences. The extracted scenario-based model enables analysing the input parallel program for deadlock-freedom, boundedness and real-time temporal properties. The programming and extraction techniques are demonstrated for the CAL actor language [23]. CAL is employed by the ISO/IEC standardization for the Reconfigurable Video Coding (RVC) MPEG standard. The extraction framework is implemented in an openly available CAL compiler [2] and interfaced with the SDF3 [86] dataflow analysis toolset. Case studies are presented for multimedia and wireless radio dataflow networks to show the applicability of the model extractor.

1.2.3 Formal Temporal Analysis

MPSoC platforms for embedded streaming applications commonly comprise heterogeneous resources that are shared between multiple applications under different scheduling policies. These applications have strict real-time constraints such as throughput and end-to-end latency. It is crucial to guarantee that such constraints are satisfied at all operating conditions. Due to this reason, predictable system designs rely on worst-case temporal bounds; i.e. lower-bound to the worst-case throughput and upper-bound to the maximum end-to-end latency. Simulation and measurement based analysis techniques cannot guarantee worst-case temporal bounds, since it is challenging to cover all possible system behaviors. Thus, analytical techniques are often used to compute safe or *conservative* temporal bounds at design-time.

Conservativeness implies here that a computed value is always worse than, or at most the same as, the worst-case value. I.e. it is lower than or equal to the minimum throughput, and higher than or equal to the maximum latency. To avoid over-allocation of scarce MPSoC resources, a *tight* temporal bound that is close to the worst-case value is desired. Next to tightness, fast analysis techniques are also essential to enable efficient exploration of the mapping and resource allocation design-space, following the platform-based design approach. The temporal analysis problem, which we address in this thesis, is then stated as follows:

Given a dynamic streaming application that is mapped onto a shared heterogeneous MPSoC platform, how can we derive a tight lower-bound to the minimum throughput and an upper-bound to maximum end-to-end latency, which enable efficient exploration of the mapping and resource allocation design-space?

An outline of our approach is given below. Following the scenario-based modeling approach, we first isolate the different operating scenarios of a dynamic streaming application, where each scenario is modeled by a SDF graph. The possible orders of scenario executions are encoded by a FSM, as illustrated at (3) in Figure 1.4. The possible scenarios and scenario sequences can be automatically extracted from a DDN input, as mentioned in Section 1.2.2. Each scenario is individually scheduled onto the MPSoC platform, which gives rise to a *scenario mapping*, illustrated at (4) in Figure 1.3. The scheduling follows a two-level hierarchical arbitration: *inter-application* and *intra-application*. Inter-application scheduling arbitrates MPSoC resources between the different applications mapped on the platform. Intra-application scheduling arbitrates a shared resource between different actors of the same application. For inter-application scheduling, we assume the minimum resource each application is guaranteed to get is given by a *worst-case resource curve (WCRC)*. A WCRC specifies the minimum amount of resource in *service units* that an application is guaranteed to get within a given time interval. Service units can be, for example, processor cycles or interconnect transactions in bytes. For intra-application scheduling of actors, we use a static-order (SO) schedule between actors mapped on the same tile.

A scenario mapping decides actor-to-processor and channel-to-interconnect bindings. In addition, it allocates resources and constructs a SO schedule between actors that are mapped on the same processor. Given a set of scenario mappings, we follow a compositional analysis approach to derive temporal bounds. The compositional approach first analyses each scenario mapping individually. Then, the results are combined, making use of the possible orders of scenario executions, given by the FSM. A scenario mapping is analysed by constructing a characterization matrix, illustrated at (5) of Figure 1.4, that captures its timing behavior over one graph iteration. The matrix is constructed using a symbolic simulation in $(max, +)$ algebra [8]. $(max, +)$ algebra is a useful tool to analyse SDF scenarios. In self-timed execution, an actor fires as soon as all input tokens have arrived. Thus, the firing time of an actor is determined by the last arriving token, i.e. the *maximum* of the production times of all input tokens. The completion time of an actor's firing is obtained by *adding* its execution time to its start time. As a result, the overall timing behavior can be analyzed using $(max, +)$.

The resulting set of matrices, along with the FSM, are then used to analyse boundedness, worst-case throughput and maximum latency, as illustrated at (6) in Figure 1.4. The details of these analyses are presented in Chapter 5 and 7.

1.3 Key contributions

This thesis makes the following key contributions.

- An automated approach is developed to extract an FSM-based SADF model from a parallel specification of a streaming application. A key enabler is the introduction of the class of Disciplined Dataflow Networks (DDNs). DDN defines construction rules for an analysable dynamic dataflow program. This contribution has been published in [78]. (Chapter 4)

- The generalized eigenmode from $(max, +)$ algebra is used to analyse boundedness of SDF scenarios. The technique is further used to develop a generalized approach to analyse the worst-case throughput of FSM-SADF models. The generalized technique lifts existing assumptions that require scenarios to be self-timed bounded and inter-scenario synchronizations to be enforced only through initial tokens on common channels. This contribution has been published in [79]. (Chapter 5)

- Scenario-based modeling and automated model extraction are respectively demonstrated for LTE baseband and RVC-MPEG-4 SP video decoder. These contributions have been published in [80] and [78]. (Chapters 3 and 4)

- An approach is developed for a matrix characterization of a scenario mapping without explicitly constructing a resource-aware model. The technique proposes embedding worst-case resource curves (WCRCs) during a $(max, +)$ symbolic simulation to characterize resource scheduling. Analysis for TDM and other schedulers under the class of $\mathcal{LR}$ servers is demonstrated, and a WCRC for Credit-Controlled Static Priority arbiter is demonstrated. Furthermore, a symbolic identification of busy times is proposed to improve the WCRT of service requests that arrive in the same busy time of a resource. The approach avoids assuming the critical instant on all requests in a busy time. The approach improves scalability and enables tighter temporal bounds. These contributions have been published in [75, 77]. (Chapter 6)

- An analytical approach is presented to derive a conservative upper-bound to the maximum end-to-end latency of an application mapping. Maximum latency is formalized in the presence of dynamically switching scenarios and then analysed under a periodic source. Applicability to aperiodic sources, such as sporadic and bursty source, is also discussed. (Chapter 7)

- The proposed analysis techniques are implemented in *SDF3* [86], a dataflow analysis tool. The model extraction approach has been demonstrated for the CAL language and implemented in the *Caltoopia* [2] CAL compiler.

1.4 Thesis Organization

The rest of this thesis is organized into seven chapters. Chapter 2 recaps basic dataflow modeling concepts and gives their formal definitions. Chapter 3 highlights dynamism of modern-day streaming applications with case-study applications. The applications are 3GPP LTE and WLAN IEEE 802.11a from wireless domain and an MPEG-4 video decoder from the multimedia domain. This chapter also presents the FSM-SADF models of the case-study applications. Chapter 4 presents an automated approach to construct analysable dataflow models, such as SDF and FSM-SADF. The approach extracts analysable models from parallel implementations, which belong to the class of Disciplined Dataflow Network (DDN). Chapter 5 presents a generalized approach to analyse the worst-case throughput of FSM-SADF. The generalization lifts existing restrictive assumptions such as self-timed boundedness and synchronizations limited to initial tokens on identical channels of scenarios. The chapter also uses the generalized eigenmode from $(max, +)$ algebra to analyse boundedness of SDF scenarios. Chapter 6 presents a new faster and tighter approach to analyse dataflow applications that are mapped onto a shared multiprocessor platform. The new approach, called Symbolic Analysis of Application Mappings (SAAM), combines symbolic simulation in $(max, +)$ algebra with worst-case resource availability curves. Chapter 7 introduces a systematic analytical approach to derive a conservative upper-bound to the maximum end-to-end latency of application mappings. Chapter 8 concludes the thesis and gives directions to future work.

CHAPTER 2

Preliminary

This chapter recaps basic dataflow modeling concepts and gives their formal definitions. It also introduces notation used in the rest of the thesis. The chapter is organized in five sections. Section 2.1 presents notational conventions. Section 2.2 briefly introduces the $(max, +)$ algebra. Section 2.3 gives formal definitions of SDF and FSM-based SADF MoCs. Section 2.4 discusses the $(max, +)$ matrix characterization of SDF scenarios. Section 2.5 introduces the CAL actor language. It also presents some motivational CAL examples that highlight the challenges of design-time of analysis of dynamic streaming applications. Section 2.6 summarizes this chapter.

2.1 Notation

We use upper-case letters (A, Θ) to denote sets and sequences, except for letters M and N that denote matrices. We use lower-case Latin letters (a) for individual elements, lower-case Greek letters $(\alpha : A \to B)$ for functions, $\mathcal{P}(A)$ for the power set of A and bar accents $(\bar{\gamma})$ for vectors. We use $|A|$ to denote cardinality or length of a set, sequence or vector. We use $\mathbb{N}, \mathbb{N}^0$ and $\mathbb{R}$ for natural numbers starting from 1, natural numbers starting from 0 and real numbers, respectively. We denote the set of real numbers extended with $-\infty$ as $\mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\}$. The set of real numbers extended with $+\infty$ and $-\infty$ is denoted as $\overline{\mathbb{R}}_{\max} = \mathbb{R} \cup \{+\infty, -\infty\}$. Exceptions to these conventions will be explicitly stated whenever used.

2.2 Max-Plus Algebra

This section presents basic $(max, +)$ algebra notation used in this thesis. For elements $a, b \in \mathbb{R}_{\max}$, $(max, +)$ algebra defines $a \oplus b \stackrel{\text{def}}{=} \max(a, b)$ and $a \otimes b \stackrel{\text{def}}{=} a + b$. In this paper, we use the standard max and addition notation for the sake of readability. For any element $a \in \mathbb{R}_{\max}$, $\max(-\infty, a) = \max(a, -\infty) = a$ and $a + -\infty = -\infty + a = -\infty$. The algebra is extended to vectors and matrices as explained in the following subsections.

2.2.1 Vectors

For $n \in \mathbb{N}$, $\underline{\mathrm{n}} \stackrel{\text{def}}{=} \{1, 2, \cdots, n\}$. An n dimensional vector is an element of the set $\mathbb{R}^n_{\max}$. For vector $\bar{\gamma} \in \mathbb{R}^n_{\max}$, the entry at row $i \in \underline{\mathrm{n}}$ is denoted as $[\bar{\gamma}]_i$. For $c \in \mathbb{R}_{\max}$, $\mathbf{u}[c] \in \mathbb{R}^n_{\max}$ denotes a vector that has c in all of its entries; i.e. for any $i \in \underline{\mathrm{n}}, [\mathbf{u}[c]]_i = c$. In addition, scalar to vector addition and multiplication are given as $[c + \bar{\gamma}]_i = c + [\bar{\gamma}]_i$ and $[c\bar{\gamma}]_i = c[\bar{\gamma}]_i$, respectively.

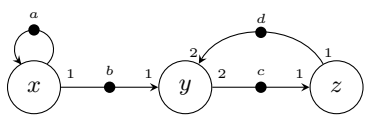
Given vectors $\bar{\gamma}, \bar{\theta} \in \mathbb{R}^n_{\max}$, we have the following properties. Vector addition, subtraction and max operation are element-wise operations, i.e. $[\bar{\gamma} \pm \bar{\theta}]_i = [\bar{\gamma}]_i \pm [\bar{\theta}]_i$ and likewise $[\max(\bar{\gamma}, \bar{\theta})]_i = \max([\bar{\gamma}]_i, [\bar{\theta}]_i)$. The *norm* of vector $\bar{\gamma}$ is the maximum entry of the vector, denoted as $\|\bar{\gamma}\| = \max_i [\bar{\gamma}]_i$. For vector $\bar{\gamma}$ with $\|\bar{\gamma}\| > -\infty$, the normalized vector is denoted as $\bar{\bar{\gamma}}$, where $[\bar{\bar{\gamma}}]_i = [\bar{\gamma}]_i - \|\bar{\gamma}\|$. We write $\bar{\gamma} \preceq \bar{\theta}$ if $\forall\ i \in \underline{\mathrm{n}}, [\bar{\gamma}]_i \leq [\bar{\theta}]_i$. Similarly, $\bar{\gamma} \succeq \bar{\theta}$ if $\bar{\theta} \preceq \bar{\gamma}$. Vector dot-product $\bar{\gamma} \cdot \bar{\theta}$ is max of sums, which is analogous to sum of products of standard algebra: I.e. $\bar{\gamma} \cdot \bar{\theta} = \max_i([\bar{\gamma}]_i + [\bar{\theta}]_i)$.

2.2.2 Matrices

The set of $m \times n$ matrices is denoted as $\mathbb{R}^{m \times n}_{\max}$. Row $i \in \underline{\mathrm{m}}$ is denoted as $[M]_{i:}$ and column $j \in \underline{\mathrm{n}}$ as $[M]_{:j}$. An entry at row $i \in \underline{\mathrm{m}}$ and column $j \in \underline{\mathrm{n}}$ is denoted as $[M]_{ij}$. Given matrix $M \in \mathbb{R}^{m \times n}_{\max}$ and matrix $N \in \mathbb{R}^{n \times o}_{\max}$, matrix multiplication is defined using vector dot-products as $[MN]_{ij} = [M]_{i:} \cdot [N]_{:j}$. Similarly, matrix-vector product is given as $[M\bar{\gamma}]_i = [M]_{i:} \cdot \bar{\gamma}$. For $M, N \in \mathbb{R}^{n \times n}_{\max}$, we write $M \preceq N$ if $\forall\ i, j \in \underline{\mathrm{n}}, [M]_{ij} \leq [N]_{ij}$. Similarly, $M \succeq N$ if $N \preceq M$. Two interesting properties of matrix multiplication are *linearity* and *monotonicity*, which are rephrased in Properties 1 and 2, respectively.

Property 1 (Monotonicity)**.** *Given vectors $\bar{\gamma}, \bar{\theta} \in \mathbb{R}^n_{\max}$, if $\bar{\gamma} \preceq \bar{\theta}$, then $M\bar{\gamma} \preceq M\bar{\theta}$.*

Property 2 (Linearity)**.** *Given vectors $\bar{\gamma}, \bar{\theta} \in \mathbb{R}^n_{\max}$, matrix $M \in \mathbb{R}^{m \times m}_{\max}$ and $c \in \mathbb{R}_{\max}$, $M(c + \bar{\gamma}) = c + M\bar{\gamma}$ and $M(\max(\bar{\gamma}, \bar{\theta})) = \max(M\bar{\gamma}, M\bar{\theta})$*



set of channels: $C = \{c_{xx}, c_{xy}, c_{yz}, c_{zy}\}$
initial token labeling: $I = \{a, b, c, d\}$
initial token placement: $\iota(a) = (c_{xy}, 1), \cdots$
WCETs: $\chi(x) = 3$, $\chi(y) = 1$, $\chi(z) = 1$

Figure 2.1: Example SDF graph

2.3 Dataflow Models of Computation

Timed dataflow models of computation (MoCs) are often used for design-time analysis of stream-based embedded applications. A dataflow model is a directed graph that consists of *actor* nodes and FIFO buffer *channels*. Such graphs are worst-case abstractions of parallel programs that comprise multiple concurrent tasks. An actor abstracts from the implementation details of a task in terms of its maximum computational requirement, i.e. the worst-case execution time (WCET), and inter-task synchronization interface, i.e. input-output data rates. Each actor has a set of *input ports* and a set of *output ports*. Actors communicate by sending data *tokens* through their ports.

A prominent dataflow model is Synchronous Dataflow (SDF) [53]. Figure 2.1 shows an example of a SDF graph (SDFG) that consists of three actors x, y and z. The numbers on the edges indicate token production and consumption rates of ports. A SDFG actor *fires*, i.e. starts execution, by consuming from each of its input ports as many tokens as the port rate. After a certain delay, given by the actor's WCET, it produces at each of its output ports as many tokens as the port rate. Therefore, SDF actors have fixed port rates that do not change between firings. In SDF schematics, black dots represent initial tokens available in channels at the beginning of execution. Initial tokens have unique labels, as shown by the letters $\{a, b, c, d\}$ in Figure 2.1. Definition 1 formally defines SDFGs.

Definition 1 (SDFG). *A SDFG $g = (A, C, I, \chi, \rho, \iota)$ is a 6-tuple, comprising a set A of actors, a multiset $C \subseteq A \times A$ of channels, a set I of initial tokens of channels, the WCET of actors $\chi : A \to \mathbb{N}^0$, the source and destination port rates of channels $\rho : C \to \mathbb{N} \times \mathbb{N}$, and initial token placement $\iota : I \to C \times \mathbb{N}$.*

The collection of the minimum number of non-zero actor firings that restores the graph back to its initial token distribution is called an *iteration*. The number of firings of each actor in one iteration is given by the *repetition vector*. E.g. the repetition vector of Figure 2.1 is $\{(x, 1), (y, 1), (z, 2)\}$, which implies that actors x and y each fire once, and actor z fires twice. A SDFG is called *consistent*, as defined in Definition 2, if it has such a repetition vector. Consistency is a necessary condition for a deadlock-free execution of a SDFG [53].

Definition 2 (Repetition Vector and Consistency)**.** *The repetition vector of a SDFG $g = (A, C, I, \chi, \rho, \iota)$ is denoted as $\nu : A \to \mathbb{N}$. It specifies the collection of the minimum number of non-zero actor firings that restores the initial token distribution. Each actor $a \in A$ fires $\nu(a)$ times in one iteration. The graph is said to be consistent if it has such a repetition vector.*

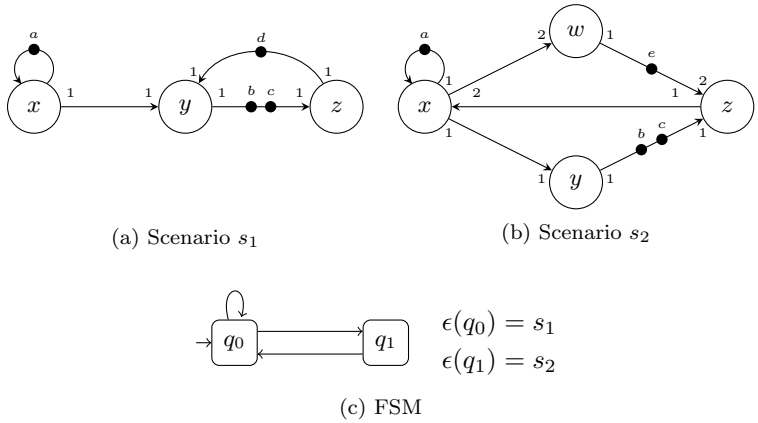
The execution of a consistent (and deadlock-free) SDFG where each actor executes as soon as it has sufficient input tokens is called *self-timed execution*. It is of special interest as it gives the maximum achievable throughput [34]. A self-timed execution that is schedulable with bounded channel storage is called *self-timed bounded* [34]. A sufficient condition for self-timed boundedness is *strong connectedness* [34]. A SDFG is said to be *strongly connected* if there is a *path* between any pairs of actors, where a path between a_0 and a_n is a sequence $\langle a_0, a_1, \cdots a_n \rangle$ of actors such that for $\forall\ 0 \leq i < n$, $(a_i, a_{i+1}) \in C$. A non-strongly connected SDFG consists of more than one *strongly connected components (SCC)*. The example SDFG shown in Figure 2.1 is not strongly connected, since there exists no path from actor y (or z) to x. Definition 3 formally defines SCCs.

Definition 3. (Strongly Connected Component) *A strongly connected component of SDFG $g = (A, C, I, \chi, \rho, \iota)$ is a maximal sub-graph of g that has a connecting path between any two actors a, b of g. The set of SCCs of g is denoted as $scc(g)$.*

SDFGs are too static to model modern-day dynamic streaming applications, such as wireless radios. These applications have varying computation and communication characteristics that change with the processed data. As a result, they go through different operating modes, called *scenarios* [89], depending on the input stream. However, the possible scenarios and the scenario sequences for input streams are often known at design time. Definition 4 defines a finite state machine (FSM) on infinite words that captures all possible scenario sequences for a given set of scenarios.

Definition 4 (Finite-state machine)**.** *Given a set S of scenarios, a finite state machine f on S is a 4-tuple $f = (Q, q_0, T, \epsilon)$. Q is a set of states, $q_0 \in Q$ is an initial state, T is a transition relation, $T \subseteq Q \times Q$, and ϵ is a scenario labeling, $\epsilon : Q \to S$.*

When an application operates at a given scenario, its characteristics mostly remain static. Hence, a SDFG can be used to effectively model and analyze it. A dynamic dataflow modeling approach, based on SDFG scenarios and their FSM is referred to as *FSM-based Scenario-aware Dataflow (FSM-SADF)* [28, 89], as defined in Definition 5. In the rest of this thesis, we use the terms scenario and SDFG interchangeably.



(a) Scenario s_1

(b) Scenario s_2

(c) FSM

Figure 2.2: Example FSM-based SADF dataflow model

Definition 5 (FSM-SADF)**.** *A FSM-based Scenario-aware Dataflow (FSM-SADF) model is a pair (S, f). S is a set of scenarios and f is an FSM on S.*

Figure 2.2 shows an example of an FSM-SADF model that has two scenarios s_1 and s_2. In the FSM, state q_0 is labeled with scenario s_1 and q_1 with scenario s_2. This FSM encodes infinitely many scenario sequences. An example scenario sequence is $\langle s_1, s_2, s_1, s_1, s_1, s_2, \cdots \rangle$. The execution of this scenario sequence begins with the execution of scenario s_1 for one iteration. At the end of the iteration, the initial tokens $\{a, b, c, d\}$ of the scenario return back to their original locations, but with different production times. The next scenario, s_2, begins its execution from the production times of these initial tokens. This way synchronization is enforced between consecutive scenarios. This is further discussed in the next section.

2.4 $(max, +)$ Characterization of a Scenario

The execution of a scenario is a timed simulation of the executions of its actors. For instance, the repetition vector of scenario s_1 of Figure 2.2 is $\nu(x) = 1, \nu(y) = 1$ and $\nu(z) = 1$. The completion time of an iteration can be obtained from the production times of the latest tokens at the end of the iteration. This collection of tokens is the same as the initial tokens $\{a, b, c, d\}$, as mentioned earlier. This is because the initial tokens of scenario s_1 return back to their original locations at the end of the iteration, but with different production times. A *time-stamp vector* $\bar{\gamma} \in \mathbb{R}^n_{\max}$ is used to record the production times of initial tokens after each iteration. Each initial token has exactly one entry in this vector.

The $(max, +)$ algebra [8] is a useful tool to compute the production times of tokens. In self-timed execution of a scenario, an actor fires as soon as all input tokens have arrived. Thus, the firing time of an actor is determined by the last arriving token, i.e. the *maximum* of the production times of all input tokens. The completion time of an actor's firing is obtained by *adding* its execution time to its start time. As a result, the overall timing behavior of a self-timed execution of a SDFG can be analyzed using $(max, +)$ expressions.

We show this with an example. Scenario s_1 of Figure 2.2 has four initial tokens: $\{a, b, c, d\}$. We represent the time-stamp vector at the end of iteration $k \geq \mathbb{N}^0$ of scenario s_1 using symbolic variables as $\bar{\gamma}_k = [t_a, t_b, t_c, t_d]$. The repetition vector of scenario s_1 equals $[1, 1, 1]$. Hence, in a single iteration, each actor fires once. Assume the actors have the following WCETs: $\chi(x) = 1, \chi(y) = 3, \chi(x) = 2$.

The only input token that actor x consumes is token a. Hence, the firing time of actor x is the availability time of token a, i.e. t_a. It then completes its firing after its WCET. This implies that after the completion of actor x, the production time-stamp of token a becomes $t'_a = t_a + \chi(x) = t_a + 1$. This can also be written as a $(max, +)$ vector dot-product as $t'_a = [1, -\infty, -\infty, -\infty] \cdot \bar{\gamma}_k$. This is because the vector dot-product is max of sums (cf. Section 2.2.1) and it evaluates to $t_a + 1 = \max(1 + t_a, -\infty + t_b, -\infty + t_c, -\infty + t_d)$.

The firing of actor y consumes a token produced by actor x on channel c_{xy}. This token has the same time-stamp as t'_a. Actor y also consumes token d from channel c_{zy}. At the end of the firing, it produces one token on channel c_{yz}, which will become token b at the end of the iteration. Therefore, the firing of y completes at $\max(t'_a, t_d) + \chi(y)$. Thus, the new token b becomes available at $t'_b = \max(t'_a + 3, t_d + 3) = \max(t_a + 4, t_d + 3)$. In vector dot-product, this is given as $t'_b = [4, -\infty, -\infty, 3] \cdot \bar{\gamma}_k$.

Similarly, after the firing of actor z, which consumes token c from channel c_{yz}, we get $t'_d = t_c + 2 = [-\infty, -\infty, 2, -\infty] \cdot \bar{\gamma}_k$. The old token b is not consumed in this iteration and at the end of the iteration this token becomes token c, due to the shifting in the FIFO. As a result, $t'_c = t_b = [-\infty, 0, -\infty, -\infty] \cdot \bar{\gamma}_k$.

Collecting the production times of the four initial tokens at the end, the time-stamp vector at the end of the $(k+1)^{th}$ iteration is given as $\bar{\gamma}_{k+1} = [t'_a, t'_b, t'_c, t'_d]$, where

$$\begin{aligned} t'_a &= [1, -\infty, -\infty, -\infty] \cdot \bar{\gamma}_k, \\ t'_b &= [4, -\infty, -\infty, \quad 3] \cdot \bar{\gamma}_k, \\ t'_c &= [-\infty, 0, -\infty, -\infty] \cdot \bar{\gamma}_k \text{ and} \\ t'_d &= [-\infty, -\infty, 2, -\infty] \cdot \bar{\gamma}_k. \end{aligned}$$

This relationship between two consecutive iterations is conveniently expressed by the recurrence relation of Equation (2.1).

$$\bar{\gamma}_{k+1} = M \cdot \bar{\gamma}_k \tag{2.1}$$

$M \in \mathbb{R}_{\max}^{n \times n}$ is referred to as the *matrix* of the scenario. The size n of the matrix is determined by the number of initial tokens; i.e. $n = |I|$. The matrix of s_1 is given by Equation (2.2). An algorithm to compute the matrix of a scenario is presented in Algorithm 1 of [28]. The algorithm constructs the matrix from a single iteration of the scenario through symbolic simulation.

$$M = \begin{bmatrix} 1 & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 3 \\ -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & 2 & -\infty \end{bmatrix} \tag{2.2}$$

Every initial token has a distinct *index* in the recurrence relation of Equation (2.1), as given in Definition 6.

Definition 6 (Initial Token Index)**.** *Given the set I of initial tokens of a scenario, initial token index is a bijection $\zeta : I \leftrightarrow \underline{n}$, where $n = |I|$ such that for $i \in I$, $\zeta(i)$ is a row index in $\bar{\gamma}_k$ and a row/column index in M.*

The initial token index for the running example is $\zeta(a) = 1$, $\zeta(b) = 2$, $\zeta(c) = 3$ and $\zeta(d) = 4$. An entry $[M]_{\zeta(i)\zeta(j)}$ of the matrix gives the minimum timing distance between the time-stamp of token i of iteration $k+1$ and token j of iteration k; i.e. $[M]_{\zeta(i)\zeta(j)} \leq [\bar{\gamma}_{k+1}]_{\zeta(i)} - [\bar{\gamma}_k]_{\zeta(j)}$. When $[M]_{\zeta(i)\zeta(j)} = -\infty$, it implies that there is no dependency between the two initial tokens. A matrix is *regular* [43] if it has no row whose elements are all $-\infty$. A scenario cannot have an initial token that has no dependency with any of the initial tokens (including itself) of the graph. Thus, the matrix of any scenario is regular.

The time-stamp vector of a self-timed bounded scenario becomes periodic after a finite number of transient iterations [28]. Thus, the normalized vector (cf. Section 2.2.1) of the time-stamp vector repeats itself after a finite number of iterations in the periodic phase. I.e. $\bar{\bar{\gamma}}_k = \bar{\bar{\gamma}}_{k-n}$, where n is referred to as the *cyclicity* factor. The average growth rate of the time-stamp vector is unique and is given by the *eigenvalue* of the matrix [28, 43].

The regularity and cyclicity properties are important concepts in worst-case throughput analysis, which is presented in Chapter 5.

2.5 CAL actor Language

Efficient exploitation of the parallelism offered by multi-core platforms is generally challenging. Sequential languages such as C/C++ are difficult to parallelize. Concurrent languages, on the other hand, expose application parallelism and enable efficient mapping onto parallel targets, such as threads and processors. CAL is an actor-oriented dataflow language [23]. Its main characterizing features are message-passing communication through FIFO buffers and state-and-data-dependent execution of actors. CAL is well-suited for programming streaming applications. In fact, CAL has been selected by ISO/IEC for the Reconfigurable Video Coding (RVC) MPEG standard. CAL can be compiled to different software (C, Java) and hardware (VHDL, Verilog) implementations [2, 3, 13].

The `Sum` actor of Listing 2.1 shows the basic constructs of a CAL actor. The actor has two input ports `in1` and `in2` and one output port `out`, all of type `int`. The functionality of a CAL actor is broken down into atomic execution units, called *actions*. Actor `Sum` has one tagged action `add`, shown at line 2 of Listing 2.1. When action `add` is fired, it consumes one token from each of its input ports and produces one token in its output port. The *port-signature* of an action specifies the number of tokens that are consumed and produced per *firing*. E.g. the port-signature of action `add` is $([1,1],[1])$ for ports ([`in1`,`in2`],[`out`]).

```
1 actor Sum() int in1, int in2 ==> int out:
2   add: action in1:[x], in2:[y] ==> out:[x + y]
3 end
```

Listing 2.1: Basic CAL actor constructs

CAL describes an application as a network of actors. Listing 2.2 shows an example of a CAL network. The network has three actor instances under the `entities` block (lines 2-5): one instance of actor `X` and two instances of actor `Y`. The connections between these actors are given under the `structure` block (lines 6-9). E.g. `x.Out1 --> y1.In` denotes a connection from output port `Out1` of actor `x` to input port `In` of actor `y1`.

```
1  network TopNetwork () ==> :
2    entities
3      x=X(); //actor instantiation
4      y1=Y();
5      y2=Y();
6    structure
7      x.Out1 --> y1.In;      //actor_name.port_name
8      x.Out2 --> y2.In;
9      y1.Out --> x.In;      //cyclic dependency to 'x'
10 end
```

Listing 2.2: Basic CAL network constructs

A CAL actor can maintain state through state variables. It can also have a finite-state machine (FSM) to encode an action `schedule` that specifies the possible orders of action firings. Along with the schedule given by the FSM, action firings are also decided by *firing rules*. A firing rule specifies conditions, in terms of actor state and input data. Each CAL action has a firing rule that must be satisfied before the action is executed. An action is termed *enabled* if its firing rule is satisfied. A firing rule may specify 1) the number of input tokens that must be available per input port; and 2) the values that input tokens and state variables must have, also known as *guard condition*. Firing rules can be both state-dependent and data-dependent, which means their evaluation may depend on state variables and input data, respectively. At a particular actor state, multiple actions may be enabled. In this case, actions are arbitrated based on a preset `priority` order.

Listing 2.3 demonstrates basic actor schedule and firing rule constructs. Actor `mc` of Listing 2.3 has three actions: `f1,f2` and `f3`. Action `f1` has higher priority than `f2` (line 18). The actor's FSM has two states: `state1` and `state2`. Actions `f1` and `f2` are associated with state `state1` (lines 21-22) and action `f3` with state `state2` (line 23). The actor has a state-variable, `payload` (line 2). Action `f1` has a guard condition that is data-dependent (line 5) and `f3` has a guard condition that is state-dependent (line 13).

```
1  actor mc() int in ==> int out:
2    int payload := 0;  //actor state
3
4    f1 : action in [ s ] ==> out: [ 1 ]
5    guard s < 0 end    //guard based on input peeking
6
7    f2 : action in [ p ] ==> out: [ 2 ]
8    do
9      payload := p;
10   end
11
12   f3 : action in [ c ] ==> out: [ 3 ] repeat 4
13   guard payload > 0  //guard based on actor state
14   do
15     payload := 0;
16   end
17
18   priority f1 > f2; end
19
20   schedule fsm state1:    //state1 is the initial state
21     state1 (f1) --> state1;
22     state1 (f2) --> state2;
23     state2 (f3) --> state1;
24   end
25 end
```

Listing 2.3: Actor schedule constructs

Initially at state `state1`, both `f1` and `f2` are possible to fire. Since `f1` has a higher priority, it is first checked for enabling. Action `f1` is fired if there is at least one input token from the input port `in` and the value of this token is less than zero, as given by the guard condition at line 5. Otherwise, action `f2` is fired. If action `f1` is fired, the FSM stays in `state1` (line 21). If, otherwise, `f2` is fired, the FSM transits to state `state2`. At state `state2`, only action `f3` is allowed to fire. Note that `f3` has also a state-dependent guard condition (line 12), which expects the state variable `payload` to be greater than zero. If this guard condition is not satisfied, the actor deadlocks, as no other action can be taken any more. If action `f3` is fired, the FSM transits to state `state1`, resuming the actor execution starting from the initial state. As it can be seen from this example, it is crucial to guarantee that a CAL actor can fire its actions indefinitely without a deadlock. This is not a trivial task, considering the expressiveness of the language that allows highly dynamic behavior. We illustrate this further with a motivational example in Section 2.5.1.

2.5.1 Motivational Example

Highly expressive actor languages such as CAL allow for data-dependent, state-dependent and/or *time-dependent* actor executions. Data-dependent and state-dependent actors take different execution paths depending on *values* of input tokens and state variables, respectively. Time-dependent actors produce possibly different results depending on the time at which input tokens are available or actions are fired [100].

The high expressiveness poses a challenge to design-time analyses such as deadlock-freedom, memory boundedness and real-time behavior. We illustrate these challenges with a simple CAL network, shown in Figure 2.3. Both actors `A` and `B` have two actions `u` and `v`. The outputs from actor `B` are feedback to set the state variable, i.e. `T fb`, of actor `A` using the execution of action `u`. All port-rates are 1, except for action `B.u`, which has the port-signature ($[1], [2]$) for ports ($[i], [o]$).

Actor `A` is deterministic, since priorities have been defined between its actions `u` and `v`. In the presence of sufficient input tokens in both input ports, the actor state (i.e. `fd`) is always updated first before the input from port j is processed. This is because action `u` has a higher priority than action `v`. However, this actor is time-dependent. If the input from port i is delayed (say due to congestion in the communication interconnect) or absent entirely (say, for instance, actor `B` deadlocks), inputs from j will be processed with the old outdated value of the state. This may result in an unexpected computational result that may further affect the rest of the network.

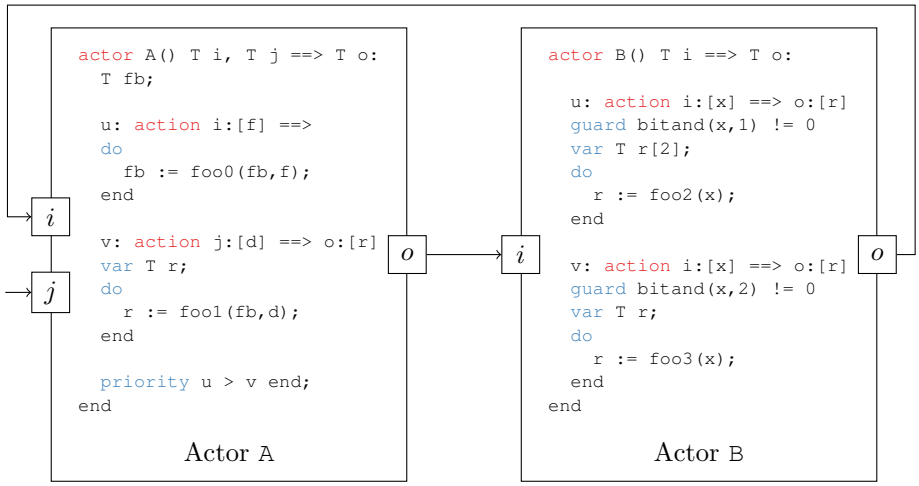


Figure 2.3: Example CAL network of two actors A and B

Actor B is data-dependent. Its actions are guarded with `bitand(x,1) != 0` and `bitand(x,2) != 0`, where `bitand` is a bitwise AND operator. These two guards neither cover the complete condition space nor are mutually exclusive. Non-mutually exclusive guards may lead to non-deterministic actor execution. For instance, this happens when $x = 3$ for these two guards. Guards that do not cover the complete condition space may lead to termination/deadlock (e.g. for $x = 4$). If actor B terminates, tokens increasingly accumulate in channel $(A.o, B.i)$, resulting in an unbounded channel. Buffer sizes are finite in reality, and hence, the network eventually deadlocks.

The challenge of design-time analysis gets worse in real-life dataflow networks, which have hundreds of such dynamic actors. To alleviate this problem, we present construction rules for a *Disciplined Dataflow Network (DDN)* in Chapter 4. The construction rules of DDN are defined in such a way that an analysable dataflow model, such as FSM-SADF and its sub-classes, can be guaranteed to be extracted from a parallel application specification. Moreover, the extraction process can also be automated. The automation enables dataflow-based system analysis to be readily available to the designer, without requiring advanced modeling expertise. In this manner, we exploit the potential of concurrent languages, such as CAL, for efficient parallel implementations of dynamic streaming applications, while still being able to 1) give guarantees on basic properties such as deadlock-freedom and boundedness, and 2) map applications onto parallel targets under real-time constraints such as throughput and end-to-end latency.

2.6 Summary

In this chapter, notational conventions and preliminary dataflow concepts are presented, which are used in the rest of the thesis. The basis of our temporal analyses in Chapters 5, 6 and 7 is the $(max, +)$ algebra, which has been summarized in Section 2.2. The temporal analyses are carried out on a FSM-based SADF analysis model that comprises a set of static SDF scenarios, which are formally defined in Section 2.3. Section 2.4 has recapped how a SDF scenario can be characterized using a $(max, +)$ matrix. The FSM-SADF analysis model is extracted from a parallel specification of a dynamic streaming application. In this thesis, the extraction process is demonstrated using the CAL actor language, whose main programming constructs were illustrated in Section 2.5. The section has also highlighted the challenges of the design-time analysis of dynamic dataflow networks with a motivational example.

CHAPTER 3

Dynamism in Streaming Applications

A challenge to the design-time analysis of present-day streaming applications is their dynamic execution behavior. These applications change their data rates and computational loads, depending on their operating mode. The intent of this chapter is to highlight such dynamic behaviors with case-study applications. The applications are selected from wireless communication and multimedia domains. Section 3.1 discusses dynamism in 3GPP's LTE and demonstrates the applicability of FSM-SADF to model and analyse LTE's baseband processing. The section also addresses the issue of modeling inter-scenario dependencies, which is revealed from LTE's FSM-SADF modeling. Section 3.2 briefly discusses dynamism in IEEE WLAN 802.11a and presents a corresponding FSM-SADF model. Section 3.3 highlights dynamism in multimedia applications, using an MPEG4 video decoder. Section 3.4 presents related work and Section 3.5 summarizes the chapter.

3.1 Long Term Evolution (LTE)

Long Term Evolution (LTE) is a recent standard in cellular wireless communication technologies. It aims at high bit rates: a downlink peak rate of up to 300 Mbit/s and an uplink of 150 Mbit/s [57]. The design of LTE receivers is quite complex due to the high bit rates and the resulting high workload. The complexity is further increased by dynamism (data-dependent variations) of frames. In this section, we focus on the dynamism of LTE's physical layer frames.

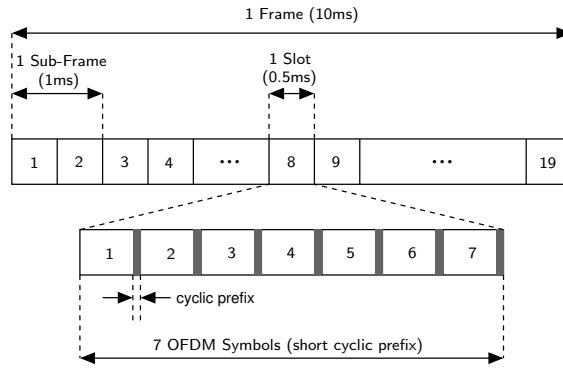


Figure 3.1: LTE frame structure for FDD

LTE uses *adaptive modulation and coding (AMC)* that dynamically adjusts modulation schemes and transport block sizes to adapt to varying channel conditions [5]. Consequently, the workload of LTE's baseband processing changes dynamically. This section presents a variation-aware dataflow model of LTE baseband processing, which captures this dynamic workload. Section 3.1.1 first discusses the sources of dynamism in the physical layer processing of LTE. Section 3.1.2 then presents the corresponding FSM-SADF model.

3.1.1 Dynamism in LTE baseband processing

There are multiple sources of dynamism in LTE baseband processing that contribute to variable computation and communication loads [71]. These include variations in channel allocation of frames, variation in length (number of symbols) of frames and variation in resource block allocations. The forthcoming discussions are limited to dynamism due to variations in channel allocations of frames. Nonetheless, the modeling concepts are equally applicable to the other types of dynamism[1].

Consider the downlink communication, which refers to the communication link from the base station to the User Equipment (UE). There are two types of LTE physical layer frame structures depending on the type of duplexing. The downlink frame structure for Frequency Division Duplexing is illustrated in Figure 3.1.

[1]Our modeling approach is to isolate a dynamic execution behavior into its static constituents and capture each static behavior with a SDF scenario. In principle, this can capture any variation in LTE that leads to varying graph structure, port-rates and executions times. For instance, our LTE model can be straightforwardly extended to handle variations in frame length, which can be either 12 or 14 OFDM symbols per sub-frame depending of the cyclic prefix used, even though our modeling covers only the case of 14 symbols per sub-frame.

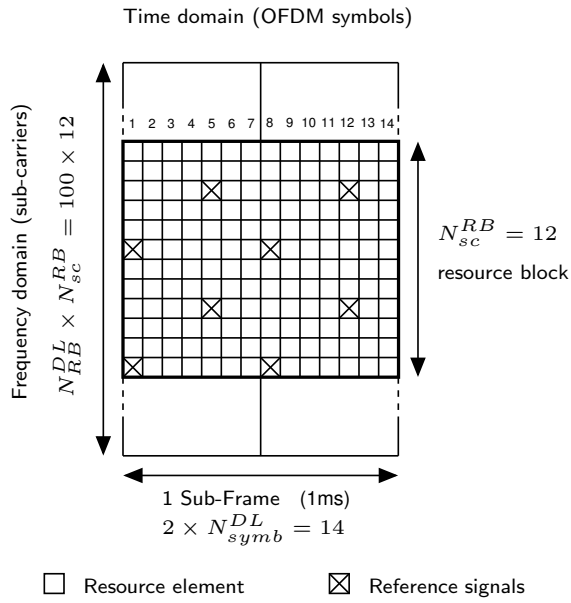


Figure 3.2: Resource grid of a sub-frame of LTE, organized into OFDM symbols (in time-domain) and frequency sub-carriers (in frequency domain)

The rest of this section details how the resource within the frame, shown in Figure 3.1, is structured and allocated to different data and control channels. The main intent of the discussion is to show that this resource allocation may vary dynamically from one frame to another. A single frame is 10 milliseconds (msec) long. It consists of 10 *sub-frames* (1msec each) and each sub-frame consists of 2 *slots* (0.5msec each). LTE employs Orthogonal Frequency Division Multiplexing (OFDM) for downlink data transmission. The transmission resource within a sub-frame is organized by a *resource grid*, as shown in Figure 3.2. The width of the resource grid (in the time domain) equals two times the number of symbols per slot, N_{symb}^{DL}. The height of the grid (in frequency domain) equals the number of OFDM sub-carriers per resource block, N_{sc}^{RB}, multiplied by the number of resource blocks per sub-frame, N_{RB}^{DL}. N_{RB}^{DL} is determined by the downlink transmission bandwidth, while N_{symb}^{DL} and N_{sc}^{RB} are determined by the OFDM subcarrier spacing and the type of OFDM cyclic prefix used (normal or extended cyclic prefix). In practice, N_{RB}^{DL}, N_{sc}^{RB} and N_{symb}^{DL} are fixed once the system is configured. From here on, we consider a bandwidth of 20MHz, a subcarrier spacing of 15KHz and normal cyclic prefix (which means 7 OFDM symbols per slot). Hence, $N_{RB}^{DL} = 100$, $N_{symb}^{DL} = 7$ and $N_{sc}^{RB} = 12$, as shown in Figure 3.2.

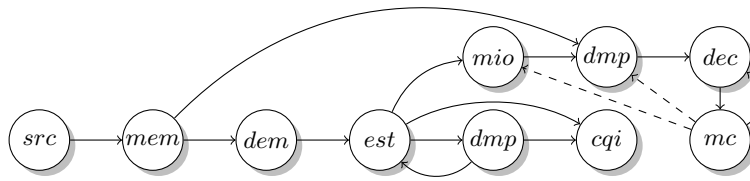


Figure 3.3: A directed task graph of LTE's baseband processing. The edges represent data-dependencies between tasks. The dashed edges indicate dependencies that exist across the processing of different OFDM symbols.

The time-frequency unit for resource allocation of the resource grid is a *resource element*. Resource elements of the resource grid are allocated to different data and control channels. Resource elements of the first OFDM symbol (the first column of the grid) are allocated to the *Physical Control Format Indicator Channel* (PCFICH) and partly to the *Physical Downlink Control Channel* (PDCCH). PCFICH contains information regarding the resource allocation of PDCCH. PDCCH can be allocated resource elements up to the third column of the resource grid. PDCCH, in turn, tells the locations of data channels, such as the *Physical Downlink Shared Channel* (PDSCH). PDSCH can be located between the second and the fourteenth columns of the resource grid.

Decoding a sub-frame consists of a number of tasks whose data dependencies are captured by a directed graph, as shown in Figure 3.3. Some major tasks of the graph include *OFDM demodulation (dem)*, *channel estimation (est)*, *multiple-input and multiple-output summation (mio)*, *OFDM demapping (dmp)* and *channel decoding (dec)*. The input-output data granularity of these tasks is an OFDM symbol (a column of the resource grid), that is about 4800 bytes. Hence, these tasks have to be carried out for each of the 14 symbols that constitute a sub-frame.

However, the functionality, execution time and data rates of tasks vary depending on the type of channel allocated to the symbol. For instance, task *dec* has a different execution time for symbols that carry a control channel than a data channel. Task *dec* takes in the worst-case 192 time-units per symbol to process a control channel, while it takes in the worst-case 975 time-units for 13 symbols allocated to a data channel, which is an average of 75 time-units per symbol. In addition, its input data rate varies between 11, 12 or 13 symbols while decoding a data channel. This is because the control channel is always between the first and the third symbols, leaving the remaining symbols for data channels. Consequently, the execution time and the input-output data rates of tasks may change from one execution to the other. This gives rise to the dynamic behavior of the application.

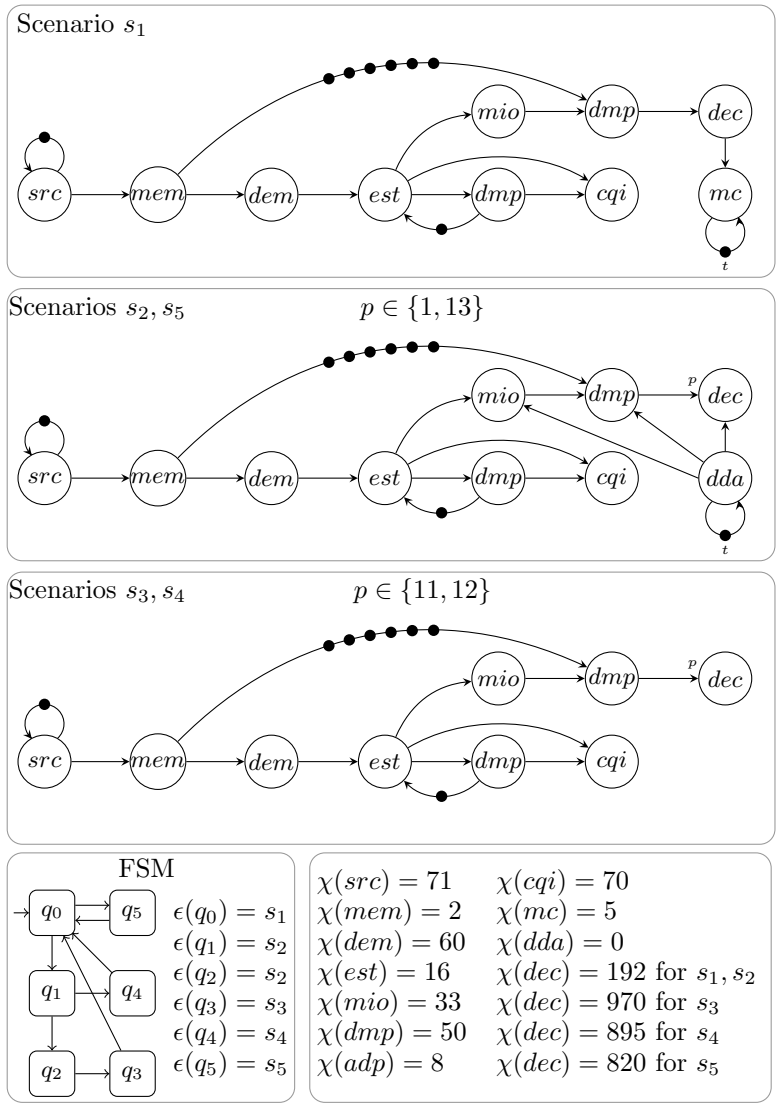


Figure 3.4: FSM-based SADF model of LTE baseband processing. The model has five scenarios. All port rates are 1, except for the input port of *dec*, which is indicated by the parameter p. The value of p equals $1, 11, 12$ and 13 tokens for scenarios s_2, s_3, s_4 and s_5, respectively.

3.1.2 FSM-SADF Model of LTE

We identify five different modes of operation of LTE, depending on the type of symbol it is processing. Each mode defines a scenario in the FSM-SADF model, shown in Figure 3.4. All port rates are 1, except for port p.

Scenario s_1 models the decoding of the first symbol, which has the control format channel (PCFICH) and part of the control channel (PDCCH). The *mode controller (mc)* actor of s_1 determines the scenario sequence to decode the remaining 13 symbols. The three possible sequences are: 1) executing s_5 to decode all the 13 symbols for the data channel (PDSCH), 2) executing s_2 to decode the second symbol for the control channel (PDCCH), followed by s_4 to decode the remaining 12 symbols for the data channel (PDSCH), and 3) executing s_2 twice to decode the second and third symbols for the control channel (PDCCH), followed by s_3 to decode the remaining 11 symbols for the data channel (PDSCH).

The scenario sequence required to decode a given sub-frame is determined by actor *mc* of s_1. Hence, the execution of actor *mc* should be completed before actors *mio, dmp, dec* of scenario graphs s_2 and s_5 start execution. The other actors in s_2 and s_3 can start execution ahead of the completion of actor *mc*. The dashed edges in the directed task graph of Figure 3.3 indicate those actors that have dependencies with actor *mc*. The dashed edges represent data dependencies that exist across scenarios: from actor *mc* of s_1 to actors *mio, dmp* and *dec* of s_2 and s_5. We refer to such types of data dependencies that exist between scenarios as *scenario dependencies*, which is further elaborated as follows.

Scenario dependencies of two scenarios are enforced through the set of initial tokens they have in common, as discussed in Section 2.3. This is because the firing times of actors that consume these common initial tokens are determined by the production times of the initial tokens in previous iterations. Hence, to enforce scenario dependencies correctly, all data dependencies between iterations should be captured through common initial tokens of scenarios. However, it is not always possible to model data dependencies using common initial tokens unless the channels that carry common initial tokens exist between scenarios. We illustrate this with the LTE example.

According to the FSM-SADF model of Figure 3.4, there are three possible sub-frame types and hence, three different scenario sequences to decode a sub-frame: $1 := \langle q_0, q_5 \rangle$, $2 := \langle q_0, q_1, q_4 \rangle$ and $3 := \langle q_0, q_1, q_2, q_3 \rangle$. The type of a sub-frame is detected by scenario s_1. At the end of scenario s_1, the mode controller (mc) actor dispatches the type of sub-frame to be decoded. This requires some kind of synchronization such that certain sub-frame-specific actors of successive scenarios (i.e. s_2 and s_5) do not fire before the sub-frame type is detected. Specifically, there are dependencies from actor *mc* of s_1 to actors *mio*, *dmp* and *dec* of s_2 and

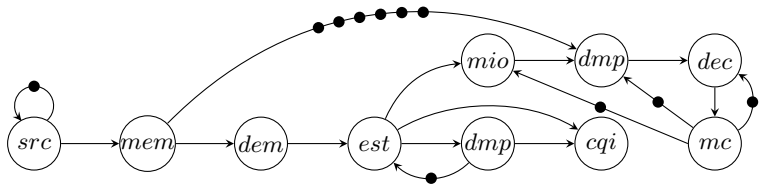


Figure 3.5: A conservative SDFG of LTE's baseband processing

s_5. These dependencies cannot be captured through initial tokens on common channels, since actor *mc* is not active in scenarios s_2 and s_5.

In such cases, we model scenario dependencies through *initial-token labeling*, which identifies a token in different scenarios using its identifier (label). This may require introducing *data-dependency actors* that carry the dependency token. In Figure 3.4, this synchronization is modeled by the initial token t and the data-dependency actor (*dda*) in scenarios s_2 and s_5. The data-dependency actor is a SDF actor, whose WCET is set to 0 and has a self-edge. The self-edge has initial token t that carries the completion time of actor *mc*. In this manner, when a scenario transition is taken from s_1 to s_2 or to s_5, actors *mio*, *dmp* and *dec* cannot start execution before the finishing time of *mc* in s_1. Inter-scenario synchronization through initial-token labeling is discussed in detail in Chapter 5, which also explains why certain initial tokens can be absent in some scenarios.

3.1.3 Conservative SDF Model

A conservative static dataflow model, such as SDF, abstracts from such dynamic behaviors for the sake of analysability. It models an application with a static graph that captures the worst-case behavior across all scenarios. However, such an abstraction may lead to overly pessimistic temporal bounds, as shown next with the SDF model of LTE.

All tasks of Figure 3.3, except *dec*, have fixed execution times and input-output token rates. Thus, the SDF modeling effort simplifies to finding a fixed execution time for task *dec* and its input port rate. The requirement for the selection of these two parameters is that the production time of tokens by actor *dec* must be conservative to (not earlier than) the actual production time of data by task *dec*. Symbols that carry control channels have to be decoded as soon as they arrive. This requires the port rate p to be 1 and $\chi(dec) = 192$. This configuration also ensures that the decoding of a data channel, which is carried out in a chunk of 11, 12 or 13 symbols, is also conservative at a sub-frame level. The resulting SDFG model is shown in Figure 3.5, where all port rates equal to 1.

Due to the static nature of the SDFG, the execution time of *dec* is fixed to $192\mu sec$ for all symbols, although it is on average $75\mu sec$ for data channels (in the worst-case $975\mu sec$ for 13 symbols). In addition, actor *mc* is executed for every symbol, even though it is only needed for the first symbol. As a result, the timing analysis of the conservative SDFG gives a pessimistic throughput of 2.6×10^{-4} sub-frames per μsec, while the refined FSM-SADF analysis, based on Figure 3.4, gives a worst-case throughput of 4.3×10^{-4} sub-frames per μsec[2].

3.2 IEEE WLAN 802.11a baseband processing

Figure 3.6 shows the FSM-SADF model of WLAN 802.11a baseband processing. This model is derived from the discussion in [59]. WLAN packets arrive sporadically (with a certain minimum time interval) and the decoding consists of 4 scenarios. The scenarios are *Synchronization* (s_1), *Header decoding* (s_2), *Payload decoding* (s_3) and *Cyclic-redundancy checking* (s_4). The FSM also has four states, which correspond to the four scenarios. All port-rates of the scenarios are 1.

Once a packet is detected, scenario s_1 executes repeatedly until synchronization succeeds. Then, scenario s_2 decodes the packet header to determine the size of the payload, which may vary from 1 to 256 OFDM symbols, each with a length of $4\mu sec$. After header decoding, scenario s_3 is executed as many times as the number of OFDM symbols. The FSM approximates this conservatively[3] by allowing an arbitrary number of payload symbols, through the self-transition on state q_2. Finally, scenario s_4 preforms a cyclic redundancy check (CRC). If CRC is successful, an acknowledgment packet must be sent within $16\mu sec$ of the reception of the last OFDM symbol. This time guard of $16\mu sec$, known as the Short Intra-Frame Spacing (SIFS), specifies a real-time latency requirement.

Actors *da*1 and *da*2 are data-dependency actors, which are introduced to model scenario dependencies, as discussed in Section 3.1.2. At the end of header decoding (s_2), several demodulation parameters need to be communicated to actor *pdem* of scenario s_3, which performs demodulation of payload symbols. This scenario dependency from scenario s_2 to scenario s_3 is modeled by actor *da*1. Similarly, actor *da*2 enforces a scenario dependency between scenario s_3 and scenario s_4.

[2]Throughput is normally given in *iterations per time-unit (ipt)*. For the sake of comparing the SDF and FSM-SADF analyses, we have converted the *ipt* results to *sub-frames per time-unit*. For the SDF graph, it requires 14 iterations per sub-frame, since each iteration processes one OFDM symbol. For the FSM-SADF, we have conservatively assumed the longest sequence, i.e. $\langle s_1, s_2, s_2, s_3 \rangle$, which has 4 iterations of different scenarios to decode a sub-frame.

[3]This implies that the FSM also specifies additional scenario sequences, which have more than 256 executions of scenario s_3 consecutively. As a consequence, the corresponding temporal analysis on the FSM is also conservative.

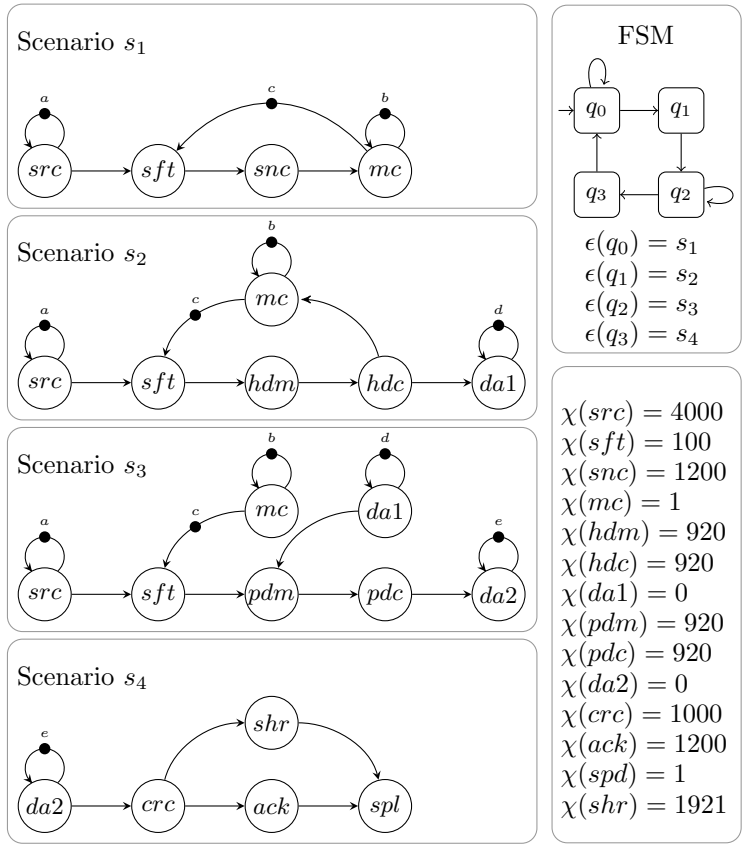


Figure 3.6: FSM-based SADF model of WLAN baseband processing

3.3 RVC-MPEG4 Simple Profile video decoder

Reconfigurable Video Coding (RVC) is an MPEG initiative that aims at providing a model to specify MPEG standards at system-level [14]. The initiative has standardized a subset of CAL, named RVC-CAL, for specifying reference software for Functional Units (FUs) (i.e. actors), which form a multitude of video codecs. This section discusses dynamism in the MPEG-4 Simple Profile video decoder (RVC-MPEG4 SP). The MPEG-4 standard defines the syntax of an MPEG-4 compliant bitstream, which can be used for coding video for different usecases. A particular decoder implementation may selectively support a number of subsets, or profiles, of the standard. The most basic profile is called Simple Profile (SP), which supports the decoding of simple rectangular video. Figure 3.7 shows a dataflow network of RVC-MPEG4 SP decoder.

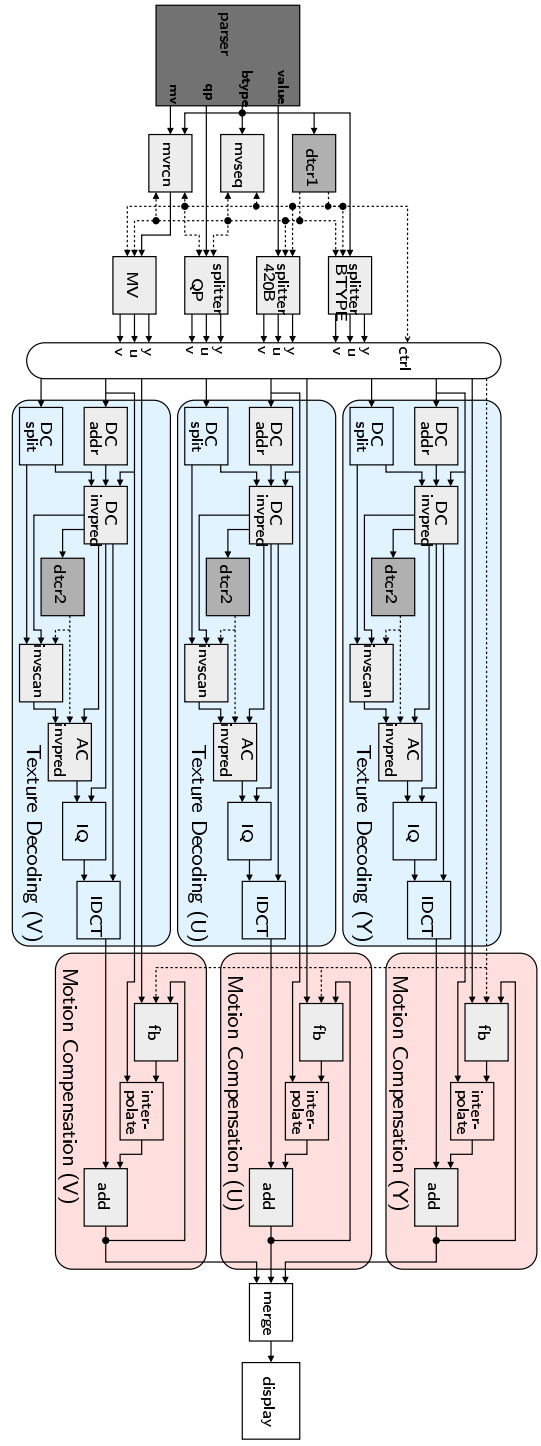


Figure 3.7: A dataflow network of RVC-MPEG4 Simple Profile (SP) video decoder

The dataflow network of Figure 3.7 has four main parts: the parser, a luminance component (Y) processing path, and two chrominance component (U, V) processing paths [46, 58]. The main functional engines of each path are the texture decoding and motion compensation blocks. The blocks are themselves sub-networks of multiple atomic actors. The parser analyses the input bitstream and extracts key parameters (such as frame size and macroblock type) that are needed to properly configure the rest of the network, depending on the decoded video frame. The three processing paths can run concurrently, as they are independent of each other and do not share any information. The processed blocks from each path are then merged to form the decoded video frame, which is fed to the display. We discuss below the decoding process in more detail to spotlight the different sources of dynamism of the decoding stages.

The input to the decoder is an MPEG-4 encoded bitstream. The decoding process performs the video encoding steps in a reverse order. There are two ways to encode a video frame in an MPEG-4 SP: an I-frame or a P-frame. An I-frame is encoded without any reference to other frames. Encoding an I-frame begins with partitioning the frame into a series of 16×16 non-overlapping colored pixel blocks called macroblocks. Each macroblock consists of six 8×8 monochrome pixel blocks, where four of them are for the luminance/brightness (Y) channel and the other two are for chrominance/color (U and V) channels (i.e. a 4:2:0 color subsampling). Each macroblock is encoded through three main processing units: discrete cosine transform (DCT), Quantization and variable-length coding (VLC). I-frame encoding is similar to a JPEG still image encoder, with only slight implementation detail differences. It only considers spatial correlations, based on information within the current video frame. A P-frame, on the other hand, exploits temporal redundancies to achieve efficient video compression. This is because consecutive video frames are significantly similar, except for changes induced by moving objects within the frames. Thus, a P-frame is a motion compensated frame, which is encoded by making use of the difference between the current frame and a previous frame (a reference frame). In this case, a residual error or difference frame is generated, which is then spatially coded similar to the encoding of an I-frame.

The computational workload of decoding a frame varies dynamically due to variations in video frame encoding. I-frames, for instance, do not have motion estimated macroblock, while P-frames do have. I-frames and P-frames also have their own variants, depending on different processes such as AC prediction, AC coding and motion estimation. For instance, different types of P-frames may result from variations in the motion estimation process. The motion estimation process examines the reference frame in the neighborhoods of the input macroblock for a closely matching macroblock. The displacement between the input macroblock

and the point where the best match is found is treated as the motion vector. Then, the motion vector and the error macroblock are both encoded. If, however, the block in the reference frame is an exact match, only the motion vector is transmitted. Another source of dynamism is also variation in the dimension of frames. This information is encoded in the bitstream at the start of a new video frame. The number of macroblocks that must be processed to decode a frame varies with the width and height of the frame. Besides frame type and dimension, the color channels (Y,U and V) also activate different regions (processing paths) of the dataflow network of Figure 3.7. Each of these channels defines a different operating scenario, even though these scenarios execute concurrently, independent of each other.

Due to the macroblock variations discussed above, the actors of Figure 3.7 have dynamic execution behavior where token production and consumption rates change between firings. The network of actors that is activated to decode a particular macroblock also changes dynamically. Constructing the FSM-SADF model of such dynamic networks is not trivial. All possible scenarios of the network need to be found and the possible orders of execution of these scenarios should also be encoded by a FSM. This requires advanced expertise in dataflow modeling and close familiarity with the application implementation. This can be time-consuming and error-prone, as the number of possible scenarios can be large in modern-day streaming applications. We address this challenge in Chapter 4 with an automated approach to extract an FSM-SADF model from a dataflow network, similar to the RVC-MPEG network of Figure 3.7.

3.4 Related Work

There are different dataflow modeling techniques that are more expressiveness than SDF and can capture different dynamic aspects of streaming applications. The list includes Khan Process Network (KPN) [48], Boolean-controlled Dataflow (BDF) and Integer-controlled Dataflow (IDF) [16], Enable Invoke Dataflow (EIDF) and Core-Function Dataflow (CFDF) [66], Variable-rate Phased Dataflow (VPDF) [96], Mode-Controlled Dataflow (MCDF) [59] and Scenario-aware Dataflow (SADF) [89].

KPN [48] describes a streaming application as a set of deterministic sequential processes that communicate through unbounded FIFO connections. No restriction is placed on the implementation of processes. Each process is permitted to read from its inputs in arbitrary order, but reading is blocking and port peeking is not allowed. Processes are deterministic, in that they always produce the same stream of output values for a given stream of input values. KPN is a superset of most other dataflow models or languages that are determinate and can be implemented

without input port peeking. KPN has been used in the literature for high-level modeling of streaming applications and system-level design-space exploration of heterogeneous embedded systems [65]. However, KPN is too expressive to analyse properties such as deadlock-freedom at design-time.

BDF and IDF [16] extend SDF graphs with variable-rate actors, which have a control port. Variable-rate actors change their input-output rates, depending on the value of a control token, consumed from the control port. Examples of such variable-rate actors include Switch, Select, Repeat-Begin and Repeat-End actors. In BDF, the value of a control token is limited to boolean values (0 and 1), while in IDF, a control token can assume integer values. In terms of expressivity, BDF and IDF are Turing-complete [16, 59], which implies memory boundedness and deadlock-freedom are undecidable. Other expressively Turing-complete dynamic dataflow models are EIDF and a restricted version of it, CFDF [66]. In EIDF, an actor may have multiple modes, where the data input-output rate is constant per mode. After an actor executes at a certain mode, it determines the set of possible next modes, which depends on the current mode and the input data consumed at the current mode. CFDF is a restriction over EIDF, where the set of possible next modes has at most one element.

The different expressively Turing-complete models, discussed above, do not allow for sufficient design-time analysability such as deadlock-freedom and boundedness, let alone real-time temporal analysis. As a consequence, they do not fit to our predictable system design methodology, discussed in Section 1.2. Instead, analysable dynamic dataflow models come close to our analysis objectives, since they intend to posses decidable properties, which may come at a price of extra restrictions on expressiveness.

One example is Variable-Rate Dataflow (VRDF) [97], which improves SDF by allowing port-rates to vary within a specified range. Its extension VPDF [96] furthermore allows actors to cycle through a number of predetermined phases, each of which may execute multiple times based on a run-time value selected from a finite interval [96]. This value may assume a value of zero, which allows modeling conditional behavior. Conservative buffer-sizes can be computed for VPDF, which meet a throughput constraint imposed by a source/sink actor [96]. VPDF focuses on local variations at actor-level and does not take into account global correlation between parameter changes. This, on the other hand, is an important behavior for better design-time analysis, since actor-level variations have a global correlation, dictated by the input data stream, as also demonstrated with the case-study applications of this chapter.

A dynamic dataflow MoC, which considers global correlations between actor-level variations, is MCDF [59]. MCDF is a restriction over IDF with the goal of supporting run-time mode switchings, along with design-time temporal analysis.

A MCDF model comprises multiple switch, select and static actors. The model has a special mode controller actor, which is a static actor that drives the control inputs of switch and select actors. Every firing of the mode controller activates a sub-graph of the complete MCDF model. Each of these sub-graphs corresponds to one mode and can be basically represented by a SDF graph. Such a graph correlates all actor firings that execute together at a global level, enabling verification of consistency and boundedness properties. In MCDF, a data-dependency across different modes, say from mode x to y, is explicitly modeled by a tunnel actor, which is a shorthand representation of a dataflow sub-network that preserves the data until mode y is executed. In our FSM-SADF modeling, an inter-scenario dependency is modeled succinctly by a single labeled initial token that carries a time-stamp from one scenario to another (cf. Section 3.1.2). The properties of a well-constructed MCDF, its temporal analysis and mapping strategies onto multi-core architectures are presented in [59]. Nevertheless, the analysis of MCDF requires conversion to homogeneous SDF (HSDF) graphs, which have a scalability problem for large graphs [35].

SADF [89] is also designed to capture several dynamic aspects of dataflow applications through the concept of global scenarios, while still ensuring analysis of correctness and long-run average and worst-case performance are decidable. It distinguishes between two types of actors: kernels and detectors. Kernels are data processing units and they consume and produce fixed amount of data tokens at a particular scenario. Detector actors configure kernel actors by sending control tokens, which determine the operating scenarios of kernels, where scenario transitions are governed by a stochastic model. The throughput analysis of SADF presented in [89] constructs a global state-space representation of the execution of scenario sequences, where transitions are at the level of individual firings of actors. This may result in a very large state-space as the graph size grows. The FSM-version of SADF, introduced in [28] and largely used in this thesis, encodes the possible orders of scenario executions using a non-deterministic FSM, thereby opening an opportunity to construct a much smaller state-space, whose transitions are at the level of scenario iterations [30].

In summary, there are multiple efforts to extend the expressiveness of static dataflow models to support dynamic aspects of modern-day streaming applications. The comparison of the different flavors of dynamic dataflow MoCs, in the strictest sense, may not be straightforward. In [16], four key aspects are mentioned: expressiveness power, compactness, ease of analysis and intuitive appeal. From the perspective of our predictable design strategy, analysability takes the central stage. Any extensions, which add to the expressiveness of a dataflow MoC, should be accompanied by the corresponding analysis techniques (that is how Chapter 5 of this thesis comes into existence). Compactness and intuitive

appeal also go together, which have significant implications when it comes to programmability and scalability. Intuitive appeal refers to how the concepts in a model are closely related to the concepts in the application being modeled [16]. MCDF [59], for instance, intends to be used as both a programming model and an analysis model. However, in its current form, where all actors are single-rate actors, it may suffer from compactness as well as scalability of the analysis. FSM-SADF, on the other hand, has a global scenario-based view of the application, which is far from the task-level view of a parallel implementation of an application. The dataflow model may also suffer from compactness as the number of scenarios increases. However, FSM-SADF allows rigorous design-time analysis, capitalizing on the analysability of SDF. In this thesis, we address the programmability and compactness issues by introducing Disciplined Dataflow Networks (DDN) as a programming model for dynamic streaming applications, while FSM-SADF is still used for all analysis purposes.

3.5 Summary

This chapter has discussed the different sources of dynamism, which contribute to the different modes of operation of modern-day streaming applications. Three applications are discussed: 3GPP's LTE (a cellular connectivity standard), WLAN 802.11a (a wireless connectivity standard) and RVC-MPEG-4 SP (a video codec standard). The sources of dynamism include variations in sub-frame and channel types of LTE, the different decoding stages and payload variations of WLAN, and variations in frame types and encoding techniques of MPEG-4 video frames. FSM-SADF models are presented for LTE and WLAN. An approach for modeling arbitrary scenario dependencies is also discussed. The challenges of manually constructing FSM-SADF models has been brought to attention through the RVC-MPEG video decoder application. Chapter 4 next addresses this challenge with an automated approach that constructs an FSM-SADF model from a parallel dataflow program.

CHAPTER 4

Disciplined Dataflow Networks

Analysing deadlock-freedom, boundedness and real-time constraints are crucial steps in the design of embedded streaming applications. To that end, the FSM-based SADF MoC isolates the individual operating scenarios and analyses the executions of the possible scenario sequences. This requires identification of all scenarios and all scenario sequences. This can be challenging because of the large number of possible scenarios in modern-day dynamic applications, as highlighted in Chapter 3. Manual construction is generally time-consuming and error-prone. This chapter addresses this challenge with an automated approach that extracts a SADF model for a class of parallel implementations, which we call *Disciplined Dataflow Network (DDN)*. DDN is designed in such a way to guarantee construction of SADF models. It also enables automating the extraction process that identifies all possible scenarios of a DDN and employs state-space enumeration to determine all possible sequences of executions of these scenarios. The approach is demonstrated for the CAL actor language and has been implemented in an openly available CAL compiler. Case studies are presented for the RVC-MPEG video decoder and WLAN 802.11a baseband processing. The chapter is organized as follows. Section 4.1 outlines the extraction approach. Section 4.2 formally introduces Dataflow Process Networks. Section 4.3 presents DDN construction rules. Section 4.4 discusses automated extraction of scenarios and scenario sequences from DDN programs. Section 4.5 presents case studies applications. Section 4.6 presents related work. Section 4.7 concludes the chapter.

4.1 Introduction

Scenario sequences enable effective design-time analysis of streaming applications [80, 88]. Analysis that does not take into account the dynamic sequences of executions of scenarios may be pessimistic and restrictive. Take, for instance, *consistency*, which is a necessary condition for deadlock-freedom and boundedness [34]. A scenario is consistent if there exists a finite number of actor executions that returns the graph back to its initial tokens distribution [53]. If scenario sequences are not considered and individual scenarios are analyzed in isolation, consistency of the application can be guaranteed only if the individual scenarios are consistent. This condition is unnecessarily constraining. With scenario sequences, it is sufficient to show that all scenario sequences in cycles of the FSM are consistent. E.g. consider a FSM with a single cycle, whose scenario sequence is $\langle s_1, s_2 \rangle$. Consistency of the application is guaranteed if this sequence is consistent; i.e. if executing scenario s_2 after scenario s_1 restores the original tokens distributions. This does not require s_1 and s_2 to be individually consistent [29, 31, 89]. A similar argument holds for *boundedness*, which guarantees an implementation within a bounded memory [34, 79].

Identifying scenario sequences also enables compositional real-time temporal analysis, such as worst-case throughput [30]. The analysis is more accurate than approaches that do not consider scenario sequences. This eventually leads to resource savings, since real-time requirements can be guaranteed at a lower resource allocation. In [88], a 66% gain in resource allocation has been reported for a video decoder application, due to the refined analysis using scenario sequences.

To exploit these benefits, a scenario-based analysis model should be constructed from the application implementation. However, this step has a number of challenges. First, the analysis model abstracts from implementation details such as how scenario switching is decided. Hence, the types of parallel implementations for which such a model can be constructed is unknown. Second, the validity of abstraction of the analysis model must be verifiable. Third, modern-day streaming applications have a large number of possible scenarios, which makes manual model construction unattractive. It is time-consuming, error-prone and requires constant revisions to maintain consistency with changes of the application.

This chapter presents an automated approach to the model construction problem. The input to the extraction process is a parallel implementation of the application, written in a concurrent language. The extraction technique is largely language-independent, since we employ Dataflow Process Network (DPN) [54] to characterize a parallel implementation of the application. DPN has been introduced to give a common denotational semantics to concurrent languages. A DPN is a network of actors that communicate by message-passing through FIFO

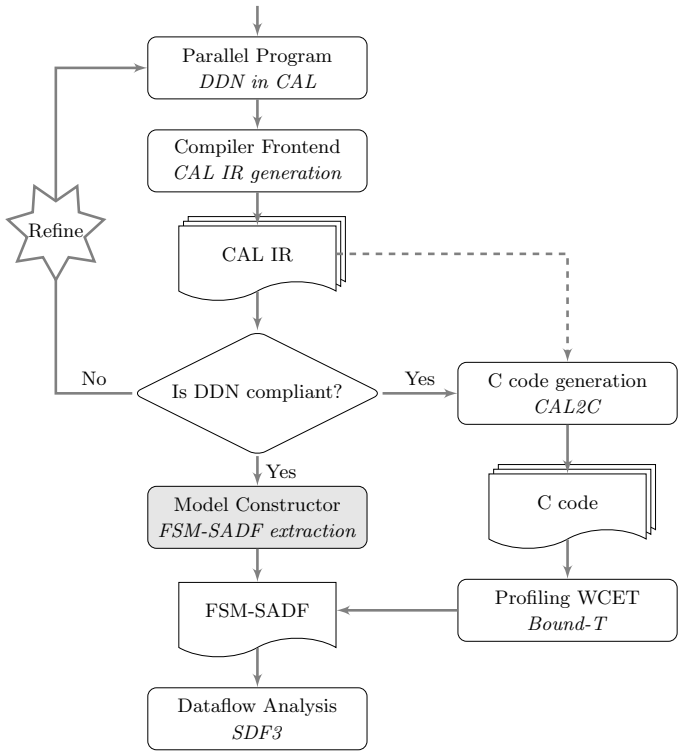


Figure 4.1: Automated scenario sequence extraction framework

buffers. Each actor has a set of different *firings*. Each firing consumes and produces a fixed number of data *tokens*. Executions of the firings are controlled by *firing rules* that specify the conditions for the execution of these firings. These conditions may be *data-dependent* and *state-dependent*, i.e. they may depend on values of input tokens and on actor state. Thus, a DPN actor may have data-dependent token production and consumption rates.

DPNs are, in general, expressively Turing-complete, and hence, it is not possible to guarantee construction of a scenario-based analysis model for arbitrary DPNs. We therefore introduce a class of parallel implementations, which we call *Disciplined Dataflow Network (DDN)*, for which construction of a scenario-based analysis is guaranteed to be possible. Moreover, the construction process is automated. The goal of DDN is to define construction rules that enforce a well-defined structure on the *control flow* that determines scenarios of a parallel implementation. To that end, DDN differentiates between *detector* and *kernel* actors, following the spirit of SADF [89]. Detectors are the initiators of variations

in dynamic network behaviors, while kernels are the followers. To keep models analysable, DDN restricts the data and state dependencies of actors. For instance, it restricts the state-dependency of kernel actors to a finite set of states and their data-dependencies to control tokens from detector actors. Compliance of an input program with such construction rules can be automatically checked.

The main contribution of this chapter is an automated extraction framework that 1) defines features of a parallel implementation, through DDN, for which construction of a scenario-based analysis model can be guaranteed, 2) identifies all possible scenarios of a DDN and extracts their SDF graphs and 3) derives all possible sequences of executions of these scenarios through state-space enumeration and constructs a finite-state machine (FSM) to characterize the scenario sequences. The extracted model enables analysing the input parallel program for deadlock-freedom, boundedness and real-time temporal properties, such as worst-case throughput [30].

The programming and extraction techniques are demonstrated for the CAL actor language [23]. The demonstration setup is shown in Figure 4.1. The input is a dataflow program written in the CAL actor language. We used *Caltoopia* [2], an open-source CAL compiler, to generate an intermediate representation (IR) of the input program. The generated IR is the input to our model extractor, which is implemented in Caltoopia. The IR is also the input to *CAL2C*, a C code generator in Caltoopia. The generated C-code is profiled to derive WCETs of actors by static code analysis using the Bound-T tool [1] for the ARM7 TDMI target. Analysis of the extracted dataflow models is carried out with the *SDF3* [86] tool.

4.2 Dataflow Process Network

Dataflow Process Network (DPN) has been introduced in [54] to give a denotational semantics to dataflow languages. A DPN program, defined in Definition 7, is a network of dataflow actors that communicate by message-passing through FIFO channels. Execution of a DPN is a possibly infinite execution of its actors. The execution of a DPN actor is a sequence of atomic *firings*. Each firing consumes input *data tokens*, performs a certain computation and produces output tokens. A firing also has an associated *firing rule* that specifies what tokens must be available at the inputs for the firing to be enabled. A DPN program is in general dynamic, as it may have actors that change their data production and consumption rates between firings.

Definition 7 (DPN)**.** *A Dataflow Process Network is a tuple* (Λ, Γ) *of a set* Λ *of dataflow actors and a set* Γ *of FIFO buffer channels.*

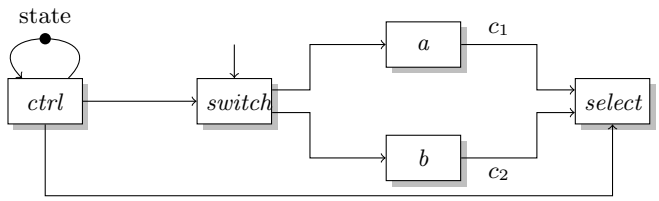


Figure 4.2: Example DPN

Let T denote the set of all tokens. A sequence of tokens is denoted as $X = [x_1, x_2, \cdots, x_n, \cdots]$ where $x_n \in T$. In a FIFO, token x_1 is at the front. We also write the n^{th} entry x_n as $X(n)$. We denote the empty sequence as $\perp$. We write $X \sqsubseteq Y$ to denote that sequence X is a prefix of sequence Y. E.g. $[x_1, x_2] \sqsubseteq [x_1, x_2, x_3]$.

The set of all finite and infinite sequences is denoted as $\mathbf{X}$. The set of n-tuples of sequences is denoted as $\mathbf{X}^n$. E.g. tuple $Z = \big[[x_1, x_2, x_3, x_4], [x_1]\big] \in \mathbf{X}^2$. We denote the cardinality of $X = [X_1, X_2, \cdots, X_n] \in \mathbf{X}^n$ as $|X| = \big[|X_1|, |X_2|, \cdots, |X_n|\big]$. The cardinality of Z is, for instance, $|Z| = [4, 1]$.

In this chapter, we exceptionally use the letter α, possibly with extra sub-and-superscripts, to denote DPN actors. For a given DPN actor $\alpha \in \Lambda$, the sets $\mathit{inports}(\alpha) = \{1, 2, \cdots, p\}$ and $\mathit{outports}(\alpha) = \{1, 2, \cdots, q\}$ denote input and output ports, respectively. Firings of the actor can be represented with a firing function that maps input tokens into output tokens, denoted as $f : \mathbf{X}^p \rightarrow \mathbf{X}^q$. Actor α may have multiple firing functions $f_{\mathit{func}}(\alpha) = \{f_1, f_2, \cdots, f_n\}$ where $f_i : \mathbf{X}^p \rightarrow \mathbf{X}^q$ for $1 \leq i \leq n$. Each firing function $f_i \in f_{\mathit{func}}(\alpha)$ has an associated *firing rule*. Hence, actor α has n firing rules $f_{\mathit{rules}}(\alpha) = \{R_1, R_2, \cdots, R_n\}$, one for each firing function.

The firing rule of a certain firing function basically specifies the amount of input tokens as well as their data values that should be available per input port for the firing function to be executed. E.g. consider the DPN network shown in Figure 4.2, which has five actors. Actor *select* has two different firing functions, say f_1 and f_2. The firing function f_1 reads only from connection 1 (c_1) and f_2 reads only from connection 2 (c_2). The execution of the actor is a series of these two firing functions (e.g. $\langle \cdots, f_1, f_1, f_2, f_1, f_2, f_2, \cdots \rangle$), where one of them is selected at a time. The selection between these two firings is decided by the value of a control token received from actor *ctrl*. The firing f_1 is selected if there is at least one token on c_1 and a token of value 1 is sent from the controller actor. Likewise, firing f_2 is selected if there is at least one token on c_2 and a token of value 1 is available from the controller. We define such firing rules in a more generic manner as follows.

The firing rule $R_i \in f_{rules}(\alpha)$ of the firing function f_i specifies *port rules* for each of the p input ports and is given as $R_i = [R_{i1}, R_{i2}, \cdots, R_{ip}] \in \mathbf{X}^p$. A firing may consume multiple tokens from a given input port. For this reason, port rule R_{ij} itself consists of a sequence of *patterns* that is applied to the input tokens of port j; i.e. $R_{ij} = [T_{ij1}, T_{ij2}, \cdots, T_{ijk}, \cdots, T_{ijn}]$ where the firing function f_i consumes $n \in \mathbb{N}$ tokens from input port j. Each pattern $T_{ijk} \subseteq T$ basically defines a set of tokens. A sequence of tokens, one from each pattern, must be available to make a port rule satisfied.

Example: assume the firing function f_i consumes $n = 4$ tokens from input port j, which has the port rule $R_{ij} = [\langle 6, 9\rangle, 0, *, \{t \in T \mid bitand(t) < 8\}]$. This port rule has four patterns that specify the following set of tokens.

1. $\langle 6, 9\rangle$ specifies the set of integers between and including 6 and 9.
2. 0 specifies the set with a single element, whose value is the integer 0.
3. $*$ specifies a set with a single token of arbitrary value.
4. $\{t \in T \mid bitand(t, 2)! = 0\}]$ specifies the set of tokens whose evaluation with the bitwise AND function $bitand(t, 2)$ gives a non-zero result.

There are multiple token sequences that satisfy this port rule. E.g. $[6, 0, 0, 3]$. This chapter uses only the patterns of the above four types and their combinations. The pattern $1 \cup \langle 4, 7\rangle$, for instance, specifies the set of tokens $\{1, 4, 5, 6, 7\}$.

Consequently, port rule R_{ij} defines a set $X_{R_{ij}}$ of token sequences of the form $[x_1, x_2, \cdots, x_{|R_{ij}|}]$. The port rule $R_{ij} = \perp$ is satisfied for any input sequence. We say a firing function is *enabled* if its firing rule is satisfied. The firing rule R_i is satisfied if $X \sqsubseteq X_j$ for each input port j, where $X \in X_{R_{ij}}$ and X_j is a sequence of tokens available on the channel connected to port j.

Example: For Figure 4.2, actor *select* has two firing rules $R_1 = \big[[1], [*], \perp\big]$ and $R_2 = \big[[2], \perp, [*]\big]$, where for $R_i = \big[R_{i1}, R_{i2}, R_{i3}\big]$, the port rules R_{i1}, R_{i2}, R_{i3} correspond to the inputs connected to actors *ctrl*, *a* and *b*, respectively. Hence, R_1 is satisfied if there is at least one token of value 1 from actor *ctrl* and one token from actor *a*. The status of connection c_2 is irrelevant for the firing.

We call a firing rule *data-dependent* if it has a port rule whose patterns depend on values of input tokens, such as port rules $R_{11} = [1]$ and $R_{21} = [2]$ of actor *select*. We call an actor *data-dependent* if it has a data-dependent firing rule. A DPN actor may have *state* that is local to the actor. Actor state can be represented by a *state* token on a *self-edge* that runs from an actor back to itself. Actor *ctrl* of Figure 4.2 has such a self-edge to model its statefulness [54]. We call a firing rule *state-dependent* if it has a port rule on the input connected to the self-edge, whose patterns depend on the value of the state. We call an actor *state-dependent* if it has a state-dependent firing rule. State and data dependencies allow us to implement dynamic actors whose input and output rates vary between firings.

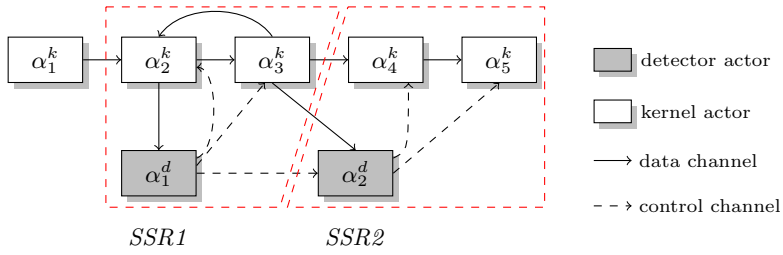


Figure 4.3: Example DDN

4.3 Disciplined Dataflow Networks

A DPN program of a streaming application processes an input stream of data objects, such as packets and frames. Processing a data object requires a sequence of firings of different actors, which we refer to as execution path. The possible execution paths should be known to analyse a DPN for properties such as boundedness and deadlock-freedom.

Firing rules of a DPN actor are determined by the actor's state and its input tokens. Therefore, the sequence of firings of an actor (i.e. its *control flow*) is decided locally. Lack of correlation between the control flows of actors poses a challenge to identify the possible execution paths. In this section, we introduce a set of construction rules that are intended to enforce some global correlation between the control flows of actors to allow for an automated analysis. We refer to a DPN program that complies with these construction rules as Disciplined Dataflow Network (DDN). Section 4.3.1 gives an overview of DDN.

4.3.1 DDN Overview

The rules of DDN enable constructing an analysable DPN. The DDN rules restrict state and data dependencies of actors to state variables and input tokens that 1) are of type integer and 2) can only assume a finite number of values. A DDN is constructed from two types of actors: *detectors* and *kernels*. Detector actors control the firings of kernel actors by means of the data tokens communicated from the detector to the kernel. All data-dependent firing rules of kernel actors are driven by output tokens of detector actors. Figure 4.3 illustrates an example DDN. The DDN has two detector actors (α_1^d and α_2^d) and five kernel actors (α_i^k for $1 \leq i \leq 5$). Outgoing channels of detector actors are by definition *control channels* and all other channels are *data channels*. A detector actor and the set of kernel actors it controls form a *Scenario Synchronous Region* (SSR). Figure 4.3 shows two SSRs: *SSR1* of detector α_1^d and *SSR2* of detector α_2^d.

The figure shows that a detector actor may control other detector actors. DDN, however, allows only hierarchical detector dependencies, which means cyclic-dependencies, for instance from α_2^d back to α_1^d, are not allowed. Table 4.1 presents example firing functions for the actors of Figure 4.3. In the table, an order for the ports of an actor is given by specifying an order for the actors, with which it is connected to. Moreover, we reused our notation of patterns of firing rules (Section 4.2) to input and output token sequences of firing functions.

Example: actor α_1^k has only one firing function $f_1^{k_1}(\left[\ \right], \left[[*]\right])$. The port order $[\ ], [\alpha_2^k]$ indicates that the actor has no input ports (i.e. the $[\ ]$ part) and has one output port connected to actor α_2^k. When this firing is taken, the actor reads no input tokens but, at the end of the firing, it produces a single token at the output port. The output token can have any arbitrary value, as indicated by the $*$ wildcard character. Similarly, the firing $f_1^{k_2}(\left[[1], [*], [*]\right], \left[[*], [*]\right])$ of actor α_2^k is taken if there is at least one token on each of its three input ports, where the token from the first input port has the value of 1, as encoded by the input part $\left[[1], [*], [*]\right]$. At the end of the firing, the actor produces one token on each of its two output ports, as given by the output part $\left[[*], [*]\right]$.

The construction rule of detector actors

1. allows arbitrary state and data dependencies, but
2. restricts the values of their output tokens to a finite set of integers.

The construction rule of kernel actors

1. reduces the state-dependency to a finite number of states that can be captured by a finite-state machine, and
2. restricts data-dependencies to input tokens from detector actors.

These rules guarantee that a scenario-based analysis model can be constructed through an automatic process. Figure 4.4 shows the extracted FSM-SADF model of Figure 4.3. It shows four extracted scenarios that capture all possible execution paths and an FSM that characterizes all possible scenario sequences.

Example: when the firing $f_1^{d_1}(\left[[*]\right], \left[[1], [1], \perp\ \right])$ of detector actor α_1^d is taken, it produces tokens of value 1 to actors α_2^k and α_3^k, but sends no token to the other detector actor α_2^d. As a result, the second detector actor is not activated and neither are actors α_4^k and α_5^k, which are in the SSR region of α_2^d. The resulting scenario is Scenario s_1 of Figure4.4. The scenario has four SDF actors a, b, c and d_1, which correspond to the active DDN actors α_1^k, α_2^k, α_3^k and α_1^d, respectively. The input-output data rates of each SDF actor are derived from the input-output token sequences of the firing function, which is enabled in its respective DDN actor. The scenario extraction process is presented in further detail in Section 4.4.1 and 4.4.2.

DDN actor	**SADF actor**	**Port order [inputs],[outputs]**	**Firings**
α_1^k	a	$[\,], [\alpha_2^k]$	$f_1^{k_1}([\,], [[*]])$
α_2^k	b	$[\alpha_1^d, \alpha_1^k, \alpha_3^k]$, $[\alpha_1^d, \alpha_3^k]$	$f_1^{k_2}([[1], [*], [*]], [[*], [*]])$, $f_2^{k_2}([[\langle 2,3\rangle], [*], [*]], [\,\bot, [*]])$
α_3^k	c	$[\alpha_1^d, \alpha_2^k]$, $[\alpha_2^d, \alpha_2^k, \alpha_4^k]$	$f_1^{k_3}([[1], [*]], [[*], \bot, \bot\,])$, $f_2^{k_3}([[2], [*,*,*]], [\,\bot, [*,*,*], [*,*]])$, $f_3^{k_3}([[3], [*]], [[*], [*], [*]])$
α_4^k	d	$[\alpha_2^d, \alpha_3^k]$, $[\alpha_5^k]$	$f_1^{k_4}([[1], [*]], [[*]])$, $f_2^{k_4}([[2], [*]], [\,\bot\,])$, $f_3^{k_4}([[3], \bot\,], [\,\bot\,])$
α_5^k	e	$[\alpha_2^d, \alpha_4^k]$, $[\,]$	$f_1^{k_5}([[1], [*]], [\,])$, $f_2^{k_5}([[\langle 2,3\rangle], \bot\,], [\,])$
α_1^d	d_1	$[\alpha_2^k]$, $[\alpha_2^k, \alpha_3^k, \alpha_2^d]$	$f_1^{d_1}([[*]], [[1], [1], \bot\,])$, $f_2^{d_1}([\,\bot\,], [[2,2,2], [2], [2]])$, $f_3^{d_1}([\,\bot\,], [[3], [3], [3]])$
α_2^d	d_2	$[\alpha_1^d, \alpha_3^k]$, $[\alpha_4^k, \alpha_5^k]$	$f_1^{d_2}([[2], \bot\,], [[1], [1]])$, $f_2^{d_2}([[3], [t\|foo(t)]], [[2], \bot\,])$, $f_3^{d_2}([[3], [t\|!foo(t)]], [\,\bot, [3]])$

Table 4.1: Example firings of DDN of Figure 4.3

We give next a BNF-like representation of DDN programs. Braces $\{\cdots\}$ refer to "zero or more of the enclosed". *'⟨(ddn) firing rule⟩' → 'firing function'* says *'firing function'* is executed if *'(ddn) firing rule'* holds. The construction *do if ... fi od* refers to a repetitive operation of selecting and executing a firing among the enclosed set of firings. The construction *par ... rap* refers to parallel execution. Section 4.3.2 next discusses ddn firing rules and the construction rules of kernel actors. Section 4.3.3 discusses the construction rules of detector actors.

- ⟨firing⟩ ::= ⟨firing rule⟩ → "firing function"
- ⟨ddn firing⟩::= ⟨ddn firing rule⟩ → "firing function"
- ⟨detector actor⟩ ::= do if ⟨firing⟩{, ⟨firing⟩} fi od
- ⟨kernel actor⟩ ::= do if ⟨ddn firing⟩{, ⟨ddn firing⟩} fi od
- ⟨actor⟩::=⟨detector actor⟩ | ⟨kernel actor⟩
- ⟨DDN⟩ ::= par ⟨actor⟩{, ⟨actor⟩} rap

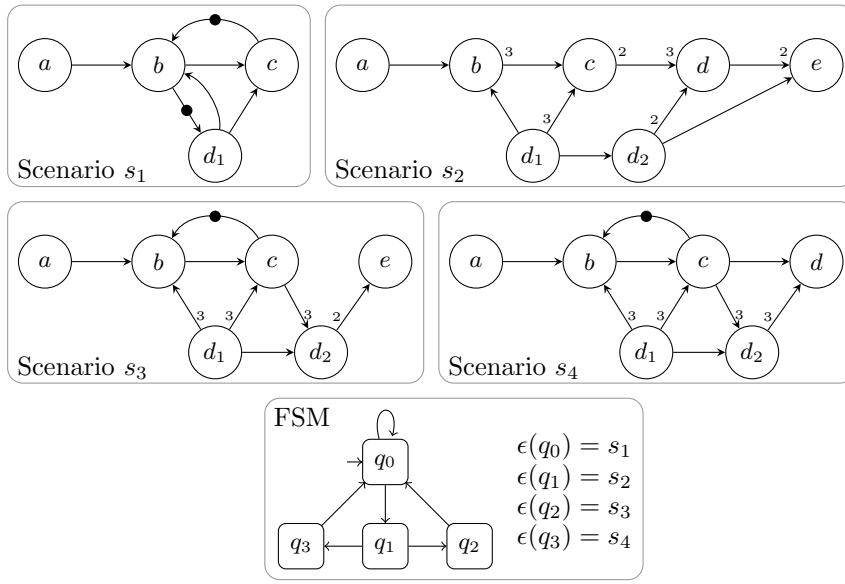


Figure 4.4: Extracted FSM-based SADF model of Figure 4.3

4.3.2 Kernel Actors

A kernel actor comprises a set of so-called *DDN firings*. Each DDN firing has a corresponding *DDN firing rule*. These rules can be both data-dependent and state-dependent, as discussed in Section 4.3.2.

DDN Firing Rules

A DDN firing rule R'_i is a firing rule where patterns that depend on values of input tokens are limited to unions of disjoint integer intervals. Each interval specifies a set of input tokens that satisfy the pattern. E.g. the firing rule $R'_i = \big[[\langle 1, 5\rangle], [\langle 1\rangle \cup \langle 5, \infty\rangle]\big]$ needs two tokens $\big[[x], [y]\big]$ to be satisfied, where $x \in \{1, 2, \cdots, 5\}$ and $y \in \{1\} \cup \{5, 6, \cdots\}$. Listing 4.1 shows an example of a CAL action that has this firing rule, shown at line 3.

```
1 actor DDNFiringRule() int in1, int in2 ==> :
2   action_tag: action in1:[x], in2:[y] ==>
3   guard x >= 1 and x <= 5, y = 1 or y > 5
4   ...
```

Listing 4.1: Example DDN firing rule

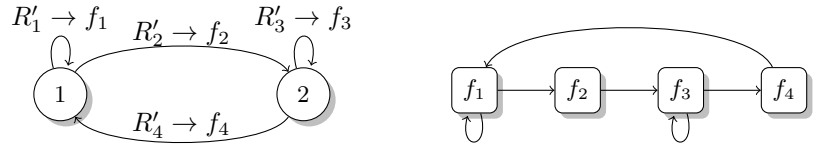
Testing patterns, which depend on values of tokens, requires input-port *peeking* in implementations. We refer to input ports on which a firing rule peeks as *input-lookaheads*. If an input-lookahead port is connected to a self-edge, which models an actor state, peeking amounts to testing state variables. An input-lookahead port, which is not connected to a self-edge, is by definition a *control port* of the firing. The DDN construction rule requires these control ports to be connected to output ports of detector actors. In the rest of the chapter, we assume a peeking depth of one, i.e. firing rules peek on only the front of input-lookahead FIFOs. This assumption simplifies the discussion of scenario extraction in Section 4.4. Generalization to arbitrarily peeking depth is straightforward. Definition 8 defines patterns of a DDN firing rule as a set of integers.

Definition 8 (DDN Firing Rule)**.** *A DDN firing rule* R'_i *of actor* $\alpha \in \Lambda$ *is a sequence* $R'_i = [R'_{i1}, R'_{i2}, \cdots, R'_{ip}] \subseteq \mathbf{X}^p$ *of port rules, for each of the* p *input ports such that for an input-lookahead port* l *where* $1 \leq l \leq p$, $R'_{il} = [t_l]$ *where* $t_l \subseteq \mathbb{Z}$. *Further,* $|t_l| \leq n$ *for some* $n \in \mathbb{N}$ *if* l *is connected to a self-edge.*

Definition 8 states that the pattern of an input-lookahead of a self-edge can only have a finite set of integer values (since $|t_l| \leq n$). This restricts the state-dependency of a kernel actor α to a finite set $states(\alpha)$ of states. Note that the actor can have any arbitrary set of state variables. The set $states(\alpha)$ specifies only the set of values (which are numbers assigned to states) that are used in DDN firing rules. State dependent behavior can be implemented with a state variable. Actor states and state transitions can also be captured by an FSM. If an FSM is not explicitly given, a fully-connected FSM can be assumed as an abstraction, as the state variable may take any value from the set $states(\alpha)$ after each firing. The FSM encodes the possible orders of firings, which we refer to as the *firing schedule* (Definition 9).

Definition 9 (Firing Schedule)**.** *A firing schedule of an actor is a directed graph, where the nodes represent firings and the edges encode the possible orders of firings.*

Example firing schedule: assume actor α has three firing rules of the form $\{[[1], \cdots], [[1], \cdots], [[2], \cdots]\}$, one for each of its firing functions, where the 1^{st} port is an input-lookahead connected to a self-edge. Thus, the set of states is $states(\alpha) = \{1, 2\}$. For the FSM given in Figure 4.5a, the corresponding firing schedule is shown in Figure 4.5b. Each state transition is labeled as $R'_i \rightarrow f_i$, implying the firing f_i is executed if the firing rule R'_i is satisfied. The firing schedule is a useful tool to detect if an actor is a *static* or *dynamic* kernel actor, as discussed next in Section 4.3.2 and 4.3.2.



(a) Actor state transitions (b) Firing schedule

Figure 4.5: Example firing schedule

Static Kernel Actor

We classify a kernel actor as static if it has a *static firing schedule*, which means none of its nodes have multiple outgoing edges. Such a schedule can execute indefinitely only if it has a cycle. Hence, a static firing schedule consists of a (possibly empty) transient sequence $f_{tran} = \langle f'_{t_1}, f'_{t_2}, \cdots, f'_{t_m} \rangle$ of firings, followed by a periodic $f_{per} = \langle f'_{p_1}, f'_{p_2}, \cdots, f'_{p_n} \rangle$ sequence of firings. A static firing schedule also implies that none of the firing rules must have data-dependent patterns, in order to ensure a non-terminating actor execution. Consequently, a static kernel actor is data-independent. A static kernel actor is categorized as an SDF actor if all of the firings in the periodic phase have the same port-signature. We use the transient firing sequence of static kernel actors to define the initial token distribution of the extracted SDF graphs, as discussed later in Section 4.4.

Dynamic Kernel Actor

We classify a kernel actor as dynamic if it has an FSM (that encodes the actor's state transitions) where each state forms a *complete set of DDN firing rules*. We say a set of DDN firing rules are *complete* if they are mutually exclusive and the union of the patterns of their input-lookaheads covers the entire condition space, as given in Definition 10. A set of DDN firing rules are *mutually exclusive* if any two firing rules of the set do not have a common tuple of control tokens that satisfies them. A complete set of firing rules ensures that exactly one firing rule gets satisfied for a given tuple of control tokens.

Definition 10 (A Complete Set of Firing Rules)**.** *Let L denote the set of input-lookaheads of the firing rule R'_i of the i^{th} firing function. Let $t_l \subseteq \mathbb{Z}$ denote the set of tokens defined by the pattern of the port rule $R'_{il} = [t_l]$ of $l \in L$. Furthermore, let $\widehat{R}_i$ denote the set of all tuple of tokens, one per input-lookahead, that satisfy input-lookahead patterns. I.e. $\widehat{R}_i = \Pi_{l \in L} t_l$. Then, a given set R' of firing rules is complete if $\forall R'_i, R'_j \in R'$, $i \neq j$, $\widehat{R}_i \cap \widehat{R}_j = \emptyset$ and $\bigcup_{R'_i \in R'} \widehat{R}_i = \mathbb{Z}^{|L|}$.*

Example: Switch and select actors, which are common in dynamic dataflow programs, are special cases of dynamic kernel actors. These actors have a one-state FSM whose transitions form a complete DDN firing rule set. Listing 4.2 shows a CAL example of a switch actor. Every time the actor fires, it consumes a control token from input port `ctrl` and a data token from input port `in`. The data token is forwarded to output port `out1` if the value of the control token is 1 (lines 2-3), to output port `out2` if the value of the control token is 2 (lines 5-6) or to output port `out3` otherwise (lines 8-9). Therefore, the firing rules are $R'_1 = \big[[1], [*]\big]$, $R'_2 = \big[[2], [*]\big]$, $R'_3 = \big[[\langle\infty, 0\rangle \cup \langle 3, \infty\rangle], [*]\big]$. Each of the three firing rules has only one input-lookahead port, which is port `ctrl`. Therefore, $R'_{11} = [1]$, $R'_{21} = [2]$ and $R'_{31} = [\langle\infty, 0\rangle \cup \langle 3, \infty\rangle]$. The actor is complete because these three port rules have no intersection (are mutually exclusive), while their union gives the whole set of integers $\mathbb{Z}$.

```
1  actor sw() int ctrl, T in ==> T out1, T out2, T out3:
2    f1:action ctrl:[c], in:[d] ==> out1:[d]
3    guard c = 1 end
4
5    f2:action ctrl:[c], in:[d] ==> out2:[d]
6    guard c = 2 end
7
8    f3:action ctrl:[c], in:[d] ==> out3:[d]
9    guard c != 1 or c != 2 end
10 end
```

Listing 4.2: Example switch actor

4.3.3 Detector actors

A detector is a DPN actor whose output tokens are of type integer and take values from a finite set. It has a finite number of firings and each firing has an associated firing rule. These firing rules can be data and state dependent, and do not have to be necessarily DDN-compliant. DDN does not check the completeness of such firing rules; although it allows for testing whether a firing of a detector is enabled by other detectors.

Listing 4.3 shows an example of a dummy detector actor, written in CAL, with three firings (i.e. actions `f1,f2` and `f3` at lines 4,7 and 12). As far as the model extraction is concerned, these three firings should not be necessarily mutually exclusive, but should completely cover the condition space to guarantee termination-free execution of the detector. Our extractor assumes termination-free execution of detectors, which can be tested through advanced techniques, such as abstract interpretation, in a separate step [99].

```
actor detector() int in ==> int out1, int out2:
  int payload := 0;

  f1 : action in [ d ] ==> out1: [ 1 ]
  guard d < 0
  end // end of action 'f1'

  f2 : action in [ d ] ==> out1: [ 2 ], out2:[ payload ]
  // 'bitand' refers to a bitwise AND operation.
  // It limits the value of 'payload' to only 0 and 4.
  do
    payload := bitand(d, 4);
  end // end of action 'f2'

  f3 : action ==> out1: [ m ] repeat 4
  // Line 24 declares a local array variable
  // 'm' of size 4. Each entry of the array is
  // initialized to 3. All four entries are
  // sent out of the output port 'out1' at the
  // end of the firing of action 'f3'.
  var int [4] m := [3 : for int i in 1 .. 4]
  do
    payload := 0;
  end // end of action 'f3'

  priority
    f1 > f2;
  end // end of priority

  schedule fsm state1:
    state1 (f1) --> state1;
    state1 (f2) --> state2;
    state2 (f3) --> state1;
  end // end of schedule
end
```

Listing 4.3: Example detector actor

Listing 4.3 also shows that a detector actor may have multiple output ports; i.e. ports `out1` and `out2`. Such output ports are by definition *control ports* of the detector if they are connected to the input-lookaheads of dynamic kernel and/or other detector actors. In this case, each firing of the detector actor produces a tuple of control tokens, one from each control port. For instance, when action `f2` is taken, it produces the control token 2 on port `out1`, and either token 0 or 4 on port `out2`. Hence, the possible tuple of control tokens for action `f2` are $(2, 0)$ and $(2, 4)$. Note that multiple control tokens of the same value are allowed per port per firing. For instance, in action `f3`, output port `out1` is a multi-rate port, where all the 4 control tokens have the same value of 3. The tuple of control tokens produced per detector firing allows us to automatically identify the group of firings in the SSR region of the detector, which are enabled by the firing. The group defines a scenario of the SSR region, as discussed in Section 4.4.1.

4.4 Scenario Sequence Extraction

This section presents extraction of scenario sequences from a DDN program in three sections. Section 4.4.1 presents extraction of a SDFG that captures a scenario of a DDN. Section 4.4.2 discusses the extraction of all possible scenario sequences. Section 4.4.3 analyses the algorithmic complexity of the approach.

4.4.1 Scenario Extraction

A scenario is defined by a *scenario configuration*, which is a tuple of firings, one from each detector in the network. Similar to [89], we define a Scenario Synchronous Region (SSR) as a sub-network whose actors are controlled by the same detector actor. We set the interval of scenario synchronicity to be one firing of the detector actor of the SSR. This means an SSR has a number of scenarios and in each of these scenarios, the repetition factor of the detector is one. Hence, a tuple of firings of detectors determines a scenario of the whole network.

Consider the set of detectors $D = \{\alpha_1^d, \alpha_2^d, \cdots, \alpha_n^d\}$ of a DDN. The set Ω of possible tuples of firings of detectors is given by the Cartesian product

$$\Omega = f_{func}(\alpha_1^d) \times f_{func}(\alpha_2^d) \times \cdots \times f_{func}(\alpha_n^d).$$

Each tuple $f_\Omega = (f_1^d, f_2^d, \cdots, f_n^d) \in \Omega$ defines a scenario configuration where $f_i^d \in f_{func}(\alpha_i^d)$.

Some control ports of a detector actor may produce no control token in some firings. E.g. output port `out2` of Listing 4.3 is inactive in actions `f1` and `f3`. For convenience, we introduce the symbol $\oslash$ to denote the output of an inactive control port. Thus, firing f_i^d produces a tuple of control tokens, one on each of the q control ports. These control tokens can assume a tuple of values from the set $c_{tokens}(f_i^d) \subseteq \{\mathbb{N} \cup \oslash\}^q$. Thus, f_Ω encapsulates a set $\Theta(f_\Omega)$ of tuples of control tokens,

$$\Theta(f_\Omega) = c_{tokens}(f_1^d) \times c_{tokens}(f_2^d) \times \cdots \times c_{tokens}(f_n^d).$$

Each tuple $(c_1, c_2, \cdots, c_n) \in \Theta(f_\Omega)$ of control tokens, where $c_i \in \{\mathbb{N} \cup \oslash\}$, specifies a network of enabled actor firings. This network defines an operating scenario of the DDN. The network can be captured by a SDF graph, since every firing function consumes and produces a preset fixed amount of tokens.

Example scenario extraction: The DDN of Figure 4.3 has two detector actors α_1^d and α_2^d. Each actor has three firing functions. The Cartesian product gives 9 scenario configurations. The configuration $f_\Omega = (f_1^{d_1}, f_1^{d_2})$ has only one tuple of control tokens: $(1, \oslash) \in \Theta(f_\Omega)$. $f_1^{d_1}$ produces the control token 1, which enables firings $f_1^{k_2}$ and $f_1^{k_3}$ of actors α_2^k and α_3^k. Actor α_1^k is a static kernel actor that is enabled by any configuration. The second detector does not produce any control

Algorithm 1 ExtractSDFGraph$((\Lambda, \Gamma), c_{tuple})$

ExtractSDFGraph$((\Lambda, \Gamma), c_{tuple})$
$A := \emptyset$ *//set of SDF actors*
$C := \emptyset$ *//set of SDF channels*
$\Lambda_q := \emptyset$ *//queue of DDN actors to be visited*
$\Lambda_q :=$ initialize queue of with source actors of Λ
while $\Lambda_q \neq \emptyset$ **do**
  $\alpha :=$ remove an actor from Λ_q
  if α is not visited & has enabled firing due to c_{tuple} **then**
    $f(X^p, X^q) :=$ get enabled firing of α due to c_{tuple}
    $a :=$ construct a new SDF actor based on f
    $A := A \cup \{a\}$ *//add SDF actor*
    $\mathcal{X}(a) :=$ profile firing f for WCET
    for all output ports of α, $o = 1 \cdots q$ **do**
      if sequence of port o, $X^q(o) \neq \perp$ **then**
        $\Lambda_q := \Lambda_q \cup \{$get actor connected to port $o\}$
      end if
    end for
  end if
end while

// construct SDF channels
for all channel $\gamma = (o, i) \in \Gamma$ **do**
  $f_s(X^{p_s}, X^{q_s}) :=$ get enabled firing of actor of port o
  $f_d(X^{p_d}, X^{q_d}) :=$ get enabled firing of actor of port i
  if $f_s \neq NULL$ and $f_d \neq NULL$ **then**
    $c :=$ SDF channel from actor of f_s to actor of f_d
    $\rho(c) := (|X^{q_s}(o)|, |X^{p_d}(i)|)$ *//assign port-rate*
    $f^o_{tran} :=$ set of transient firings of actor of port o
    $f^i_{tran} :=$ set of transient firings of actor of port i
    $t_{prod} := \sum_{\forall f(X^p, X^q) \in f^o_{tran}} |X^q(o)|$ *//tokens produced by transients*
    $t_{cons} := \sum_{\forall f(X^p, X^q) \in f^i_{tran}} |X^p(i)|$ *//tokens consumed by transients*
    number of initial tokens of $c := t_{prod} - t_{cons}$
    $C := C \cup \{c\}$ *//add SDF channel*
  end if
end for
return $g = (A, C, I, \chi, \rho, \iota)$

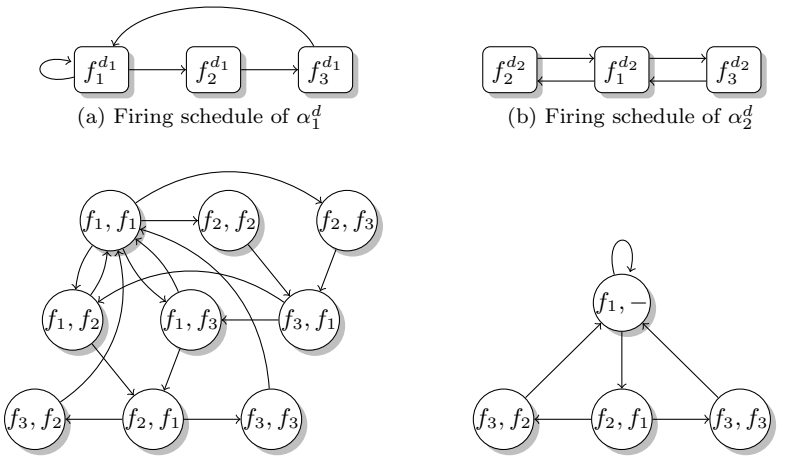
tokens, since it expects a control token of value 2 or 3 to be enabled. As a result, none of the actors in its SSR has an enabled firing. The SDFG defined by $(1, \oslash)$ is shown by scenario s_1 of Figure 4.3. The port rates of the actors of this SDFG are derived from the input-output token sequences of the firing functions, which are enabled in their respective DDN actors. For this example, all port rates are equal to 1, since the four enabled firing functions, i.e. $f_1^{k_1}, f_1^{k_2}, f_1^{k_3}$ and $f_1^{d_1}$, all have input-output token sequences of length 1.

Algorithm 1 gives a sketch of the extraction of an SDFG from DDN (Λ, Γ), given a tuple $c_{tuple} = (c_1, c_2, \cdots, c_n)$ of control tokens. It starts with initializing a queue of DDN actors Λ_q with source actors of the DDN (line 5). As long as this queue is not empty (line 6), the algorithm picks a previously non-visited actor α from the queue (line 7) and checks it if it has an enabled firing (lines 8-9). A firing of a kernel actor is enabled if its firing rule is satisfied by c_{tuple}. A firing of a detector actor is enabled if its firing function is in the scenario configuration f_Ω of c_{tuple}. An SDF actor is created based on the enabled firing function of α (lines 10-12). Then, all DDN actors that are directly connected to actor α through output ports, which have non-empty token sequences in the enabled firing, are added to the queue Λ_q (lines 13-17). The SDF channels are constructed as outlined between lines 20-30. For each connection in the DDN, the source and destination firings f_s and f_d that are enabled due to c_{tuple} are first extracted (lines 21-22). If both firings exist (line 23), a channel is created (line 24), whose port-rates are determined by the port-signatures of f_s and f_d (line 25). These port-rates can also be multi-rate (i.e. > 1). The initial tokens of the channel are set (line 26-27) by summing the tokens produced by the transient firings (cf. Section 4.3.2) of the source actor to the output port connected to the channel.

4.4.2 Extracting Scenario Sequences

Extracting all scenario configurations using the Cartesian product Ω of the set of firings of detectors is ineffective, since it may give configurations, which do not occur in any actual execution of the network. In this case, the extraction gives an FSM-SADF model, which is unnecessarily large and whose FSM is fully-connected. Consequently, the resulting analysis such as consistency and throughput would be pessimistic. This section discusses extraction of scenario configurations more accurately from local FSMs of detector actors.

We construct the possible scenario configurations through *configuration-space exploration*. The resulting configuration-space (CS) is a directed graph whose nodes and edges represent configurations and configuration transitions, respectively. The exploration is a configuration-space enumeration, based on the firing schedules of the detector actors. The exploration starts by forming the tuple of



(a) Firing schedule of α_1^d

(b) Firing schedule of α_2^d

(c) Complete configuration-space

(d) Simplified configuration-space

Figure 4.6: Construction of configuration-space

the initial firings of all firing schedules. Then, the exploration discovers new tuples (i.e. new nodes) by making a transition in each of the firing schedules. Due to hierarchical dependencies between detectors, some firings of a configuration may not be enabled. Thus, a transition is made in a firing schedule only if its firing is enabled in the current configuration. Otherwise, the firing schedule stays in the same firing until it is enabled, while other detectors make progress. The exploration terminates when no more new tuples are discovered.

Example: Assume Figures 4.6a and 4.6b are the firing schedules of detectors α_1^d and α_2^d of Figure 4.3, whose firings are given in Table 4.1. Starting from $f_1^{d_1}$ and $f_1^{d_2}$, two transitions are possible for each of the firings. This gives four possible tuples from the node (f_1, f_1) (the superscripts d_1 and d_2 are dropped for readability), as shown in Figure 4.6c. Progressing from these four tuples eventually gives the complete configuration-space of Figure 4.6c. The complete space can be reduced if we avoid paths that lead to *non-existent* and *redundant* tuples. For instance, one of the four possible tuples after the initial node $(f_1^{d_1}, f_1^{d_2})$ is $(f_2^{d_1}, f_2^{d_2})$. This tuple cannot occur, since, as we can see from Table 4.1, $f_2^{d_1}$ sends a control token of value 2, which does not satisfy the firing rule of $f_2^{d_2}$. Hence, all paths following $(f_2^{d_1}, f_2^{d_2})$ can be avoided from the exploration. The same reasoning applies also to $(f_2^{d_1}, f_3^{d_2})$ and $(f_3^{d_1}, f_1^{d_2})$. Tuples $(f_1^{d_1}, f_1^{d_2})$, $(f_1^{d_1}, f_2^{d_2})$ and $(f_1^{d_1}, f_3^{d_2})$ are redundant, since in these cases α_2^d is not controlled by detector α_1^d and the network is solely controlled by α_1^d. Removing non-existent and redundant configurations gives a simplified configuration space, shown in Figure 4.6d.

Algorithm 2 Construct configuration-space (CS) from given a set of detectors

1: ConstructCS($D = \{\alpha^{d_1}, \alpha^{d_2}, \cdots, \alpha^{d_n}\}$)
2: $(V, E) = (\emptyset, \emptyset)$ //CS is a directed graph
3: $\Omega_q := \emptyset$ //a set of configurations to be explored
4: $\Omega_q := \Omega_q \cup \{(f_1^{d_1}, f_1^{d_2}, \cdots, f_1^{d_n})\}$ //add the initial configuration
5: **while** $\Omega_q \neq \emptyset$ **do**
6: $f_\Omega :=$ remove a configuration from Ω_q
7: **if** f_Ω is not visited and is valid **then**
8: $V := V \cup \{$add a new node$\}$
9: //define the set $\Theta(f_\Omega)$ of tuples of control tokens
10: $\Theta(f_\Omega) = c_{tokens}(f_1^{d_1}) \times c_{tokens}(f_1^{d_2}) \times \cdots \times c_{tokens}(f_1^{d_n})$
11: **for all** $c_{tuple} = (c_1, c_2, \cdots, c_n) \in \Theta(f_\Omega)$ **do**
12: **if** c_{tuple} is a valid tuple **then**
13: $g =$ ExtractSDFGraph$((\Lambda, \Gamma), c_{tuple})$
14: $S := S \cup \{g\}$ //add scenario if unique
15: **end if**
16: **end for**
17: **end if**
18: **for all** f_Ω^{next} directly reachable configuration from f_Ω **do**
19: **if** f_Ω^{next} is a valid configuration **then**
20: $\Omega_q := \Omega_q \cup \{f_\Omega^{next}\}$
21: add transitions to E accordingly
22: **end if**
23: **end for**
24: **end while**
25: **return** (V, E)

Algorithm 2 shows a sketch of the CS construction from a given set of detector actors. The CS is a directed graph (line 2), where each node corresponds to a scenario configuration. The set Ω_q keeps configurations to be yet visited (line 3). The algorithm begins with the initial configuration (line 4) and searches for new configurations (lines 18-23) by making step transitions in each of the firing schedules of the detector actors. For every valid configuration, a new node is added to the CS (line 8). For each tuple of control tokens, which is defined by the configuration (lines 10-11), a scenario graph is extracted (lines 12-15) using Algorithm 1. The outgoing edges of a node are inferred from the set of directly reachable configurations (line 21). Checking for non-existent and redundant configurations and scenarios are already embedded in the exploration (lines 7,12,14,19).

The CS is a key step towards extracting an FSM-SADF model. The FSM of the extracted FSM-SADF model is constructed directly from the CS. A configuration f_Ω possibly encodes multiple tuples of control tokens, as shown in Algorithm 2 (lines 10-11). These tuples of control tokens lead to a set S_{f_Ω} of scenario graphs. Hence, multiple FSM states Q_{f_Ω} are possible for CS node defined by f_Ω such that each FSM state is labeled with a scenario graph from the set S_{f_Ω}; i.e. $\epsilon_{f_\Omega} : Q_{f_\Omega} \rightarrow S_{f_\Omega}$, where ϵ_{f_Ω} is scenario labeling. In the simplest case, each configuration has exactly one scenario graph and, therefore, translates to an FSM state. The FSM transitions directly follow the edges in the CS. An FSM transition is added between two FSM states if there is an edge in the CS between the nodes of these FSM states.

4.4.3 Complexity

Algorithm 1 consists of two loops. In the first loop, each DDN actor is tested exactly once whether it has an enabled firing. This results in a complexity of $O(|\Lambda|)$. The second loop tests each DDN channel once, and thus, has $O(|\Gamma|)$. Hence, the algorithm has a linear complexity $O(|\Lambda| + |\Gamma|)$ with the number of actors and channels of the DDN.

The model extractor is correct by construction (i.e. it finds all possible scenarios), since it enumerates all possible scenario configurations. Enumerating all configurations, however, may negatively impact the scalability of the extractor. The CS construction, sketched in Algorithm 2, has complexity $O(n)$ where n is the total number of tuples of control. n is given as

$$n = \sum_{f_\Omega \in \Omega} |\Theta(f_\Omega)|$$

where Ω is the set of all possible tuples of firings of detectors (the set of all scenario configurations) and $\Theta(f_\Omega)$ is the set of all possible tuples of control tokens of a scenario configuration $f_\Omega \in \Omega$. n increases exponentially with the number of detectors $|D|$, since the lengths of the set Ω and $\Theta(f_\Omega)$ are respectively

$$|\Omega| = \Pi_{i=1}^{|D|} |f_{func}(\alpha_d^i)| \quad \text{and} \quad |\Theta(f_\Omega)| = \Pi_{i=1}^{|D|} |c_{tokens}(f_i^d)|.$$

The control tokens $c_{tokens}(f_i^d)$ of firing f_i^d (of the i^{th} detector in f_Ω)) also increases exponentially with the number q of control ports of the detector. Thus, the CS construction has an exponential time complexity $O(k^{|D| \times q})$ with number of detectors and control ports, where k indicates the number of possible values for a control token. Constructing the FSM states and transitions (which are not shown in Algorithm 2) each require visiting every node of the CS once. Hence, their complexity is linearly to the size of the CS.

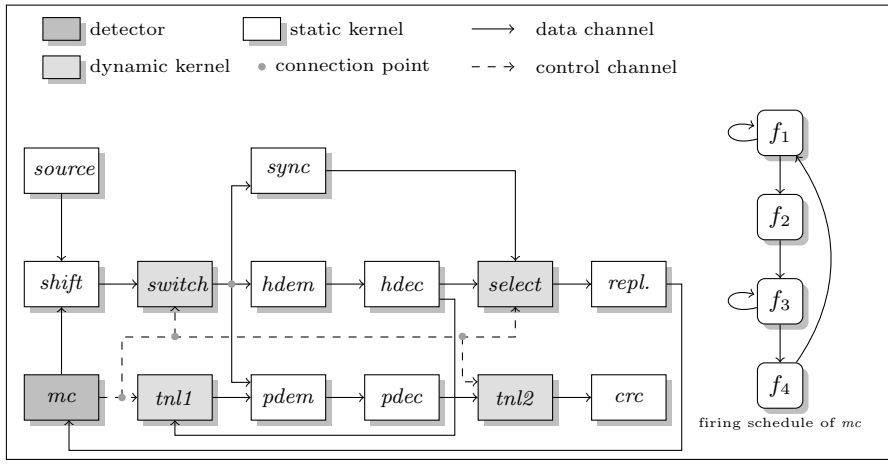


Figure 4.7: DDN program of WLAN 802.11a baseband

4.5 Case Study

This section demonstrates automated extraction of scenario sequences for WLAN 802.11a baseband processing and RVC-MPEG video decoder applications. The demonstration uses the framework presented in Figure 4.1.

4.5.1 DDN program of WLAN 802.11a Baseband

Figure 4.7 shows the DDN program of the baseband (physical layer) processing of WLAN 802.11a, based on the discussion in [59]. The DDN network is basically equivalent to the Mode-Controlled Dataflow (MCDF) model presented in [59]. The network has one detector actor (actor *mc*), four dynamic kernel actors (actors *switch, select, tnl1, tnl2*) and nine static kernel actors. Actors *tnl1* and *tnl2* are tunnel actors in MCDF, which are used to implement data-dependencies across different modes. A tunnel stores a data token produced at one mode and releases it in another mode.

The WLAN packet decoding consists of four operating modes, as discussed in Section 3.2. These modes are determined by the mode controller actor `mc`, whose CAL implementation is outlined in Listing 4.4. The actor has four actions `f1,f2,f3` and `f4` that correspond to the four operating modes. As indicated by the FSM schedule (lines 25-31), only action `f1` can be enabled at the initial state `state1` (line 25). Action `f1` tests if synchronization has succeeded. It is assumed that synchronization does not succeed at beginning, i.e.

```
1  actor mc() int from_repl ==> int to_shift, int to_control:
2    int payload := 257; //let 257 indicate an uninitialized payload size
3
4    f1: action from_repl:[data] ==> to_shift:[param()], to_control:[1]
5    guard check_sync(data) <= 0 end
6
7    f2: action from_repl:[data] ==> to_shift:[param()], to_control:[2]
8    guard check_sync(data) > 0 end
9
10   f3: action from_repl:[data] ==> to_shift:[param()], to_control:[3]
11   guard payload > 0
12   do
13     if payload = 257 then
14       payload := payload_size(data);
15     end
16     payload := payload - 1;
17   end
18
19   f4: action from_repl:[data] ==> to_shift:[param()], to_control:[4]
20   guard payload = 0
21   do
22     payload := 257;
23   end
24
25   schedule fsm state1:
26     state1 (f1) --> state2;
27     state2 (f1) --> state2;
28     state2 (f2) --> state3;
29     state3 (f3) --> state3;
30     state3 (f4) --> state1;
31   end
32 end
```

Listing 4.4: Detector `mc` of DDN of WLAN shown in Figure 4.7

`check_sync(data) <= 0` evaluates to true. This enables the firing of `f1`, which dispatches the control token 1 and sets the network in synchronization mode. Then, the scheduler transits to `state2` (line 27-28), at which either action `f1` or `f2` are enabled, depending on the result of the previous synchronization mode. If synchronization has not succeeded, action `f1` becomes enabled again. Otherwise, action `f2` gets enabled, whose firing dispatches the control token 2 and sets the network in header decoding mode. After header decoding, the scheduler transits to `state3` (line 29-30), at which either `f3` or `f4` can be enabled for firing. Action `f3` and `f4` set the network in payload decoding and cyclic redundancy check (CRC) modes, respectively. Action `f3` is fired as many times as the number of symbols (i.e. `payload > 0`). Action `f4` gets enabled when `payload = 0`, which sets the network in mode 4 for CRC. After a single firing of action `f4`, the schedule returns back to `state1` for the next packet decoding.

As shown in Table 4.2, the automated extractor identified four scenarios that correspond to these four modes. Different regions of the DDN are activated for each of these four scenarios. Since the DDN has only one detector actor `mc`, each firing f_i of `mc` determines a scenario that comprises those actor firings that are enabled by the control token produced by f_i. For instance, when `mc` executes one of its firing functions $f_1([[*]],[[*],[1]])$, it broadcasts a control token of value 1, which dictates the synchronization scenario s_1, shown in Figure 3.6, (actor *switch, select* and *repl* are not shown in the scenario).

Number of	WLAN	RVC-MPEG
Configurations (Cartesian product) $\|\Omega\|$	4	320
Configurations (after exploration)	4	32
Tuples of control tokens $\sum\|\Theta(f_\Omega)\|$	4	302
Scenario graphs $\|S\|$	4	16
FSM states $\|Q\|$	4	72
FSM transitions $\|T\|$	6	516

Table 4.2: Results of the automated model extraction

The FSM that encodes all possible sequences of executions of these four scenarios is also shown in Figure 3.6. The FSM follows the firing schedule of detector `mc`. The self-loops at states q_0 and q_2 encode the possibilities of multiple executions of s_1 until synchronization succeeds and s_3 for decoding each of the OFDM symbols in the payload, respectively.

4.5.2 DDN program of RVC-MPEG4 SP

For our case study, we used an RVC-CAL implementation of an MPEG4 Simple Profile (RVC-MPEG4 SP) video decoder. We changed all CAL guards to conform to DDN firing rules and introduced actors (which are later identified as detectors by the tool) to control these firing rules. A simplified view of the final DDN is shown in Figure 4.8. The input bitstream parser is treated as the external source, and hence, is not part of the DDN.

The DDN has two detector actors `dtcr1` and `dtcr2` that detect frame types and color channels in the input video bitstream. There are three instances of `dtcr2`, one for each of the three color channels (Y, U and V). A reduced version of the CAL implementation of detector `dtcr1` is shown in Listing 4.5. The listing shows three actions, namely `newVop, skipWH` and `mb_y`. The actor also has two more actions `mb_u` and `mb_v`, whose implementations are not shown, but their implementations are by large similar to `mb_y`.

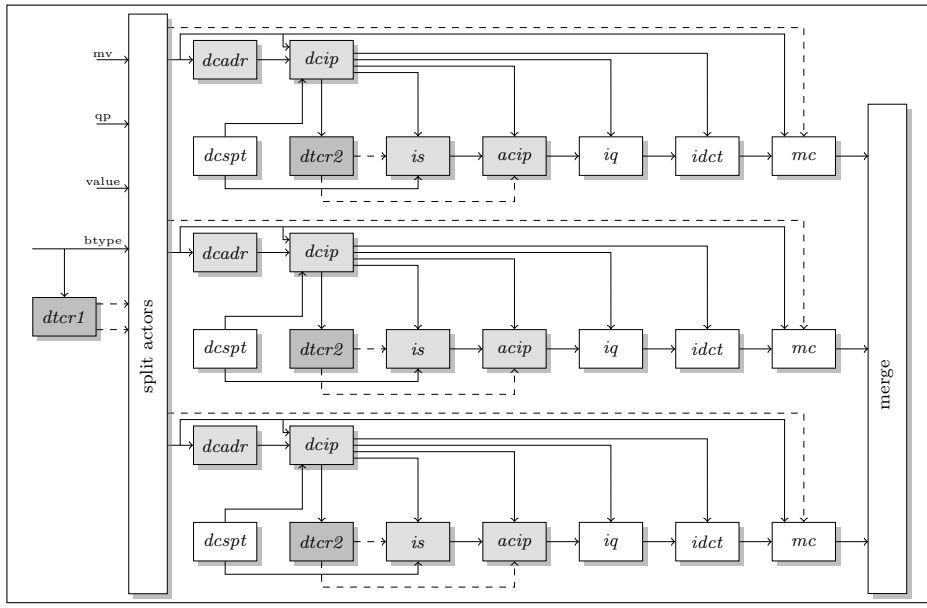


Figure 4.8: DDN program of RVC-MPEG4 SP

`dtcr1` has an input port `from_Parser` that is connected to the bitstream parser. The parser processes the input bitstream, in which the start of a new video object is marked by a three-byte sequence, named `btype`. `btype` encodes values such as `NEWVOP` (the start of a new video frame), `INTRA` (intra-frame or I-frame type), `INTER` (inter-frame or P-frame type), `MOTION` (motion prediction) and others. Thus, `btype` encodes the type of frame that is being decoded. Detector actor `dtcr1` receives `btype` from the parser and encodes it into a control token that may have 7 possible integer values; i.e. $\langle 1, 5\rangle \cup \{13, 15\}$. These control tokens are sent out through the control ports `to_In, to_Y, to_U` and `to_V` to set the operating mode of the network. For instance, when a new frame is detected, action `newVop` is fired, which sends out the control token 1. This control token gets the network ready for processing the coming macro-blocks.

The schedule of the actor (lines 42-52) shows that after a new video frame is found, the macro-blocks (MBs) of the frame are processed. Each MB comprises 6 blocks of 8×8 pixels (4 blocks for channel Y and the other two for channels U and V). Y-blocks are processed by action `mb_y`, U-blocks by `mb_u` and V-blocks by `mb_v`. These actions send out different control values, depending on the frame type. E.g. they send 3 for an intra-frame (line 24), 4 for an intra-frame with AC prediction (line 26) and 5, 13 or 15 for inter-frame (line 30) (depending on whether

```
actor dtcr1 (int ACCODED, int ACPRED, int BTYPE_SZ, int FOURMV,
             int INTER, int INTRA, int MOTION, int NEWVOP)
int from_Parser ==> int to_In, int to_Y, int to_V, int to_U, int to_C:
  int sb := 0;
  int sc := 0;

  newVop: action from_Parser: [btype]  ==>
                to_In:[sb], to_Y:[sb], to_V:[sb], to_U:[sb]
  guard bitand(btype,NEWVOP) != 0
  do sb := 1; end

  skipWH: action from_Parser: [btype] repeat 2 ==>
                to_In: [scen] repeat 2, to_Y: [scen] repeat 2,
                to_V:  [scen] repeat 2, to_U: [scen] repeat 2
  guard sb = 1 or sb = 2
  var int [2] scen
  do sb  := 2; scen[0] := 2; scen[1] := 2; end

  mb_y: action From_Parser: [btype] ==>
              to_In: [sb], to_Y:[sb], to_Comp:[sc]
  guard sc < 4, bitand(btype,NEWVOP) = 0
  do
    if  bitand( btype, INTRA ) != 0 then
      sb := 3;
      if bitand( btype, ACPRED ) = 0 then
        sb := 4;
      end
    else
      if bitand( btype, INTER ) != 0 then
        sb := 5 + bitand (btype, 15);  //take the last four bits + 5
      else
        sb := 21;
      end
    end
    sc := sc + 1;
  end

  mb_u: action ...

  mb_v: action ...

  schedule fsm newVop:
    newVop (newVop) --> skipW;
    cmd     (newVop) --> skipW;
    cmd     (mb_y)   --> y1;
    y1      (mb_y)   --> y2;
    y2      (mb_y)   --> y3;
    y3      (mb_y)   --> u;
    u       (mb_u)   --> v;
    v       (mb_v)   --> cmd;
    skipW (skipWH)   --> cmd;
  end
end
```

Listing 4.5: Detector `dtcr1` of DDN of RVC-MPEG of Figure 4.8

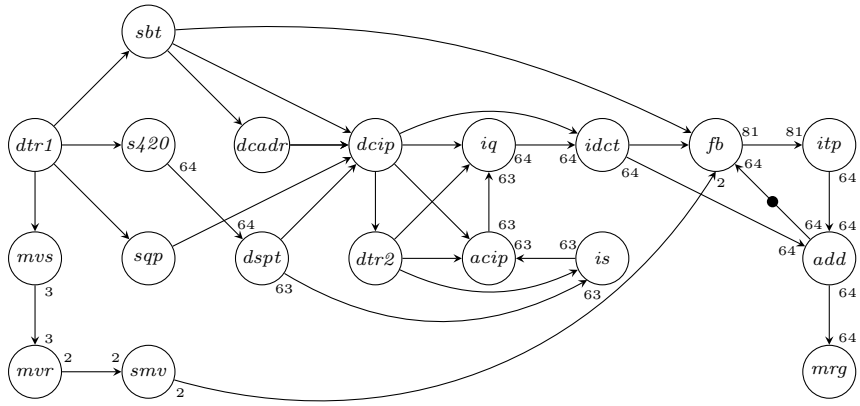


Figure 4.9: One of the extracted scenarios of DDN of Figure 4.8

AC prediction, AC coding and motion compensation are applied). In addition to this control token, actor `dtcr1` also generates another control token through port `to_C` that indicates the index (1 to 6) of the pixel block being processed. In this case, action `mb_y` sends out values $1-4$, `mb_U` the value 5 and `mb_V` the value 6.

Another dynamism in the RVC-MPEG DDN is due to the flag `AC_PRED_DIR`. The flag is sent from actor `dcip` (DC inverse prediction) to actor `acip` (AC inverse prediction) to communicate the direction of prediction. The flag also carries scan-mode information to actor `is` (inverse scan). The flag can have five integer values (-2 to 2): E.g. -2 for `NEWVOP`, -1 for an uncoded block and 0 for no-inverse-AC-prediction but zig-zag-inverse-scan. The second set of detector actors, i.e. `dtcr2`, map the flag to a control token with three possible integer values: flag value -2 to control token 1, flag value -1 to control token 2 and flag values $0, 1, 2$ to control token 3.

Every possible combination of control tokens from actor `dtcr1` and the three instances of actor `dtcr2` activate a sub-network of the DDN, and hence, define a certain operating scenario. As shown in Table 4.2, the Cartesian product of the firings of the four detector actors gives 320 scenario configurations (i.e. $f_{func}(\texttt{dtcr1}) \times f_{func}(\texttt{dtcr2})^3 = 5 \times 4^3 = 320$). The configuration-space exploration has eliminated 288 configurations, discovering that they do not occur. The remaining 32 valid configurations have 302 tuples of control tokens, which resulted in only 16 unique scenario graphs. The first two scenarios are to initialize the network, such as setting the frame width and height, when a new frame is detected in the input stream. Six scenarios correspond to processing different 8×8 Y-channel pixel blocks. The other eight scenarios are for different U and

V-channel pixel blocks. One of the scenario graphs is shown in Figure 4.9. Some connections are left out for readability and all non-indicated port-rates are 1. The tuple of control-tokens is $((15, 1), (3), \oslash, \oslash)$ from the firing of `dtcr1` and the firing of `dtcr2` in Y-channel. $\oslash$ indicates that the firing schedule of `dtcr2` is not enabled for channels U and V. The tuple encodes the processing of the first 8×8 pixel block (channel Y), which is AC coded and motion compensated.

For the WLAN and RVC-MPEG DDN networks, the run-times of the model extractor are less than a minute on a 2.8GHz Intel dual-core Linux PC.

4.6 Related Work

Dataflow Actor and Network Classification

Classification of CAL actors into known MoCs has been studied in [15, 40, 99, 103]. In [99], abstract interpretation is used to identify CAL actors with static, cyclo-static or quasi-static behavior. The interpreter executes the implementation of each actor separately, by consuming tokens whose values are unknown. The interpreter stops either when the criterion for a given MoC is satisfied or a guard condition that evaluates an unknown value is encountered. The interpreter first checks for a static (SDF) actor. The criterion is all actions must have the same input-output rates. If an actor is found not be static, a cyclo-static check follows: the criterion is an actor must have a state and a fixed number of data-independent firings that return the actor to its initial state. Lastly, an actor is checked for quasi-static behavior: it must have an FSM that branches to a finite number of transitions from the initial state. Each branch must eventually return back to the initial state, with the possibility of having cycles within a branch or across branches. Branching is solely determined by control tokens and all branching conditions must be mutually exclusive. The abstract interpreter is applied on each branch, assuming the value of the control token is known, to check whether there are a fixed number of data-independent firings that return the graph back to its initial state. The work is limited to actor-level classification, intended for optimized code generation. Hence, they do not generate a dataflow model (neither scenarios nor their sequences), as presented in this chapter, for analysing basic and temporal properties of the entire dataflow network.

[40] presents a framework that extracts statically analysable regions from CAL programs. The main purpose of the presented approach is to enable construction of quasi-static schedules for a CAL application. The basis of the methodology is to group and cluster ports together that form a SDF-like region, referred to as a Statically Schedulable Region (SSR), in the CAL network. Identifying SSRs allows to construct a static-order schedule for each SSR. This reduces the run-

time scheduling overhead as run-time decisions are only needed at SSR boundaries, resulting in a quasi-static schedule. Our work is not limited to regions and static patterns. We isolate every possible static execution behavior of the entire dynamic network and model each with a SDF model. However, we achieve this only for input programs that comply with our DDN construction rules.

In [103], a conservative classification technique is presented to identify SDF and CSDF actors from a general dataflow model. The general dataflow model is similar to a Dynamic Process Network (DPN) actor and is extracted from a SystemC code. The classification algorithm works on a *dynamic state transition diagram*. The state of an actor is determined by its state variables and FSM state. By merging the transition edges of the diagram, the algorithm finds fixed token consumption and production patterns and associates them with a (cyclo-static) state. The work also proposes a reduced state transition diagram by abstracting from the state variables and only using the FSM states, since the full state-transition diagram may be intractable. Similar to [40], the work targets only identifying static actors, with the aim of optimizing static sub-networks of a dynamic network. [15] discusses an automated approach for creating quasi-statically scheduled RVC implementations through dynamic code analysis. Accuracy of a dynamic code analysis depends on the coverage of the input streams. Thus, it cannot guarantee finding all possible execution paths. Our work employs DDN construction rules, which bound the possible program execution paths to a finite and manageable size. Thus, we avoid computationally heavy techniques such as abstract interpretation and dynamic code analysis, which suffer from path explosion. This implies that our work trades expressiveness for analysability and easier implementability.

Another key feature of our work, in contrast to all of the above works, is that we target extracting the possible orders of executions of scenarios in a form of a FSM, which enables a more accurate analysis of dynamic streaming applications.

Dataflow MoC generation

This is the first work to introduce an automated approach to construct SDF scenario sequences from a parallel application implementation. Nevertheless, automatic derivation of different dataflow MoCs from sequential code have been studied in the literature.

[49], [94], [62] and [82] transform a nested affine loop program written in Matlab/C++ into a Polyhedral Process Network (PPN) specification, which enables a systematic mapping of applications onto multi/many-core platforms. [51] introduces the LIME programming model to address the programming challenge of MPSoC platforms. It uses a “restricted C”, with certain programming constructs,

to program algorithmic blocks (actors). It also uses an XML graph description to specify dependencies between actors. The programming model can express different dataflow MoCs such as KPN, SDF and CSDF. A compiler toolchain is also demonstrated to automatically generate code for multi-core platforms. [32] extracts a SVPDF analysis model from a sequential application program, written in a domain-specific language called OIL [33]. A given OIL program is automatically parallelized into a task graph. The parallelization step extracts function level parallelism such that a task is created for each function in the input program. Synchronization statements are inserted into each task to preserve the functional behavior of the input program. A SVPDF model is then extracted from these synchronization statements.

Static models, which are considered in these works, such as SDF, CSDF and PPN cannot express dynamic streaming applications. KPN, on the other hand, is expressively Turing-complete and properties such as deadlock-freedom are undecidable. SVPDF can express variable port-rates of a dynamic application, while still allowing for design-time analysis. Our model extraction work targets FSM-SADF models, for which deadlock-freedom and boundedness properties are decidable at design-time. Unlike the sequential input codes used in the above works, we extract such models from DDN-complying parallel programs that are written in dataflow languages such as CAL. We capitalize back-end tools such as CAL2C to generate sequential C code for individual actors of such parallel programs.

4.7 Summary

The FSM-based SADF MoC enables effective design-time analysis of dynamic streaming applications. Yet, the construction of SADF models from application implementations has a number of challenges. First, the model abstracts from implementation details such as how scenario switching is decided. Hence, the types of implementations for which such a model can be constructed are unknown. Second, the validity of abstraction of the model must be verifiable. Third, manual construction is time-consuming for large number of scenarios. This chapter addresses these challenges with an automated approach that identifies and constructs all possible scenarios for a class of parallel implementations, which we call Disciplined Dataflow Network (DDN). The construction rules of DDN enable separation of scenario detection from scenario execution. Thus, a state-space enumeration can be employed to find all possible sequences of scenario executions and encode them into a finite-state machine (FSM). The extraction process requires input programs to conform to DDN. This may require certain programming effort. This effort is minimized since compliance of an input program to

DDN can be automatically tested. If a program fails the test, the unclassified actors, which are neither kernel nor detector, are reported with an indication why they failed. This facilitates debugging and improves productivity. The remaining effort is justified by acknowledging the three main benefits of DDN. DDN can be seen as an analysable programming model for adaptive streaming applications. 1) It ensures that a scenario-based analysis model can always be constructed. 2) It guarantees the adequacy of abstraction of the analysis model for deadlock-freedom, boundedness and temporal analysis, since the analysis model has the same execution semantics as the parallel implementation. 3) The process of constructing the analysis model can be automated. The model extraction approach has been demonstrated for the CAL actor language.

CHAPTER 5

Worst-Case Throughput Analysis

Wireless and multimedia embedded applications have stringent temporal constraints. The arrival and production rates of frames impose throughput requirements that must be satisfied. These applications are often dynamic and streaming in nature. FSM-SADF has been proposed to model such dynamic streaming applications, as discussed in detail in previous chapters. FSM-SADF splits a dynamic system into a set of static scenarios. Each scenario is modeled by a SDF graph. The possible scenario transitions are specified by a FSM.

An execution of FSM-SADF is an execution of the possible scenario sequences specified by the FSM. In a distributed implementation, the execution of scenarios is pipelined. As a result, multiple scenarios may be concurrently active. This poses a challenge to analyse scenario sequences. The techniques in [28, 30] analyse each scenario first individually. Then, they derive performance guarantees for a sequence compositionally. However, these techniques require two conditions. First, they assume scenarios must be self-timed bounded. A scenario is self-timed bounded if the number of tokens in every channel is bounded in a self-timed execution [34], where an actor fires as soon as all of its input data have arrived. Second, they assume all inter-scenario synchronizations (data dependencies between scenarios) are captured by initial tokens that are common between scenarios. An initial token is common between two scenarios if it is located on the same channel in both scenarios. These two conditions can be too restrictive for real-life application graphs, as illustrated by the motivational example in Section 5.1.

This chapter presents a generalized FSM-SADF analysis approach to bound the worst-case throughput. The chapter has three main contributions. 1) It presents a technique to analyse a SDF scenario that is not necessarily self-timed bounded. The technique is based on the concept of *generalized eigenmode* from $(max, +)$ algebra. Combined with a FSM, this contribution also enables to verify if a given FSM-SADF model is self-timed bounded or not. 2) It introduces *initial token labeling* to flexibly specify arbitrary scenario dependencies. 3) It presents exact and conservative worst-case throughput analysis techniques for FSM-SADFs that possibly have non-self-timed bounded scenarios.

The remainder of the chapter is organized in eight sections. Section 5.1 highlights the contributions with a simple motivational example. Section 5.2 elaborates the problem addressed in this paper. Section 5.3 characterizes a SDF scenario in terms of its self-timed components. This is followed by the analysis of a scenario and a set of scenarios in Section 5.4 and Section 5.5, respectively. Experimental results are presented in Section 5.6. Section 5.7 discusses related work and the paper concludes in Section 5.8.

5.1 Motivational Example

FSM-SADF can sufficiently express dynamic streaming applications while lending itself to design-time analysability [80]. However, restrictions on existing analysis techniques limit its applicability for many application graphs. We show a simple example with LTE baseband processing, whose FSM-SADF model is shown in Figure 3.4 and is discussed in detail in Chapter 3.1. LTE's downlink frame consists of 10 sub-frames. Each sub-frame has 14 OFDM symbols. These symbols are allocated to data and control channels. The three possible allocations are 1, 2 or 3 control channels, followed by 13, 12, or 11 data channels, respectively. The FSM-SADF model has five scenarios, which are labeled s_1 to s_5. Scenario s_1 decodes a control format indicator channel that is allocated at symbol 1. Scenario s_2 decodes a control channel that is allocated at symbols 2 and 3. Scenarios s_3, s_4 and s_5 decode a data channel that is allocated from symbol 2 to 14, 3 to 14, and 4 to 14, respectively. This results in three possible scenario sequences and hence, three different sub-frame types. The sequence for sub-frame type 1, 2 and 3 are respectively $\langle s_1, s_5 \rangle$, $\langle s_1, s_2, s_4 \rangle$ and $\langle s_1, s_2, s_2, s_3 \rangle$.

Sub-frames are decoded by executing the possible scenario sequences specified by the FSM. An example sequence is $\langle s_1, s_5, s_1, s_2, s_2, s_3, s_1, s_5, s_1, s_2, s_4, \cdots \rangle$ and its corresponding sub-frame sequence is $\langle 1, 3, 1, 2, \cdots \rangle$. The type of a sub-frame is detected by scenario s_1. At the end of scenario s_1, a mode controller (mc) actor dispatches the type of sub-frame to be decoded. This requires some kind of

synchronization such that certain sub-frame-specific actors of successive scenarios (i.e. s_2 and s_5) do not fire before the sub-frame type is detected. This synchronization is modeled by the initial token t and the data dependency actor (*dda*) in Figure 3.4. t is like a common variable (flag) that is read and written by actors *mc* and *dda*. Actor *dda* has zero execution time. Thus, in scenario transitions from s_1 to s_2 or to s_5, actor *dmp* cannot start execution before *mc* of s_1 has finished.

Modeling sub-frame detection in LTE violates the two analysis assumptions, which are mentioned earlier in the introduction. First, scenarios s_2 and s_5 are not self-timed bounded. Self-timed executions of these scenarios result in a continuously increasing accumulation of tokens in outgoing channels of actor *dda* (since *dda* has zero execution time). Second, initial token t is not a common initial token between s_1 and s_2 (s_5), as it is not on the same channel, common between these scenarios. Initial token t in s_1 and initial token t in s_2 (s_5) are two independent initial tokens. Thus, synchronization is not enforced. Consequently, existing techniques are not directly applicable to analyse the FSM-SADF of Figure 3.4.

5.2 Problem Description

This chapter studies the worst-case throughput (WCT) of FSM-SADF. The analysis of FSM-SADF is challenged by three issues. 1) In distributed implementations, such as in MPSoCs, scenario executions are pipelined. Thus, multiple scenarios are concurrently active. 2) Scenarios have data dependencies between each other. This requires inter-scenario synchronization. 3) If a scenario is not *self-timed bounded*, different components of the scenario execute at different rates of iteration. This section discusses these three issues and outlines how we address them. The main contributions of the chapter are on the second and the third issues.

5.2.1 Pipelined Execution of Scenarios

For the first issue, we follow a similar strategy as in existing approaches. The techniques introduced in [28] and [30] analyse pipelined scenario executions by executing scenario sequences, specified by the FSM. The FSM can possibly specify infinitely many state sequences. An example state sequence of the FSM in Figure 5.2 is $\tilde{q} = \langle q_0, q_1, q_2, q_2, q_2, q_0, q_1, \cdots \rangle$. The scenarios in a sequence $\tilde{q}$ are executed one after the other. The scenario of each FSM state $q \in \tilde{q}$, i.e. $\epsilon(q)$, is executed for one iteration. During the execution of a scenario, all data needed for synchronization are time-stamped with their production times. Then, the next scenario is executed, starting from these time-stamps. Therefore, data dependencies between two consecutive scenarios are captured by the time-stamps of the synchronization data, as discussed in Section 2.4.

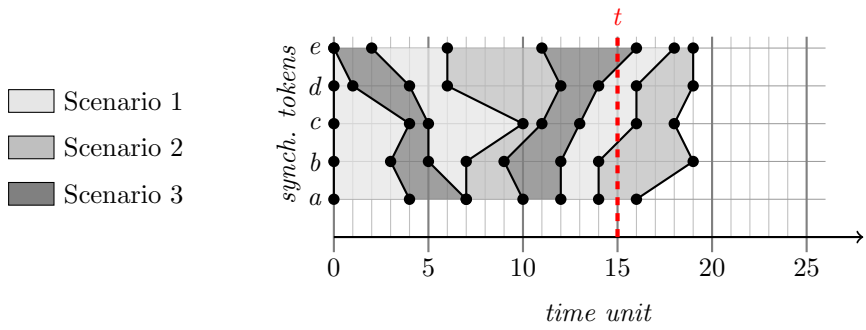


Figure 5.1: Example scenario sequence: $s_1 s_3 s_1 s_2 s_3 s_1 s_2$

Figure 5.1 illustrates an example pipelined (overlapping) execution of a sequence that involves three scenarios s_1, s_2, s_3. At time t, all the three scenarios are concurrently active. The sequence has a length of 7 iterations: $s_1 s_3 s_1 s_2 s_3 s_1 s_2$. A sequence is basically formed by concatenation of cycles of the FSM. For example, the FSM of the LTE model of Figure 3.4 has three cycles. Any scenario sequence specified by this FSM is made up of the three fragments $\langle s_1, s_5 \rangle$, $\langle s_1, s_2, s_4 \rangle$ and $\langle s_1, s_2, s_2, s_3 \rangle$, one for each sub-frame type.

We define the throughput of a sequence as the long-run average of completed *iterations per time-unit*. However, this long-run average does not necessarily exist for all sequences. Instead, it may bounce within a superior and inferior limiting bounds. We take the limit inferior as the WCT of a sequence. Then, we define the WCT of FSM-SADF as the minimum among the WCTs of all scenario sequences, as given by Definition 11.

Definition 11. (Worst-Case Throughput) *The worst-case throughput ρ of FSM-SADF (S, f) is given as*

$$\rho = \min_{\tilde{q} \in \mathcal{L}(\tilde{q})} \liminf_{k \to \infty} \frac{k}{c_{\tilde{q}^k}} \tag{5.1}$$

where $\mathcal{L}(\tilde{q})$ is the set of sequences specified by FSM f, $\tilde{q}^k$ is the first $k \in \mathbb{N}$ elements of $\tilde{q}$ and $c_{\tilde{q}^k} \in \mathbb{N}$ is the completion time of $\tilde{q}^k$.

We say an estimated WCT ρ_e is *conservative* if $\rho_e < \rho$ and *exact* if $\rho_e = \rho$. In Section 5.5, we present both exact and conservative WCT computation techniques. These techniques can also be used to derive conservative throughput bounds for sequences of finite length.

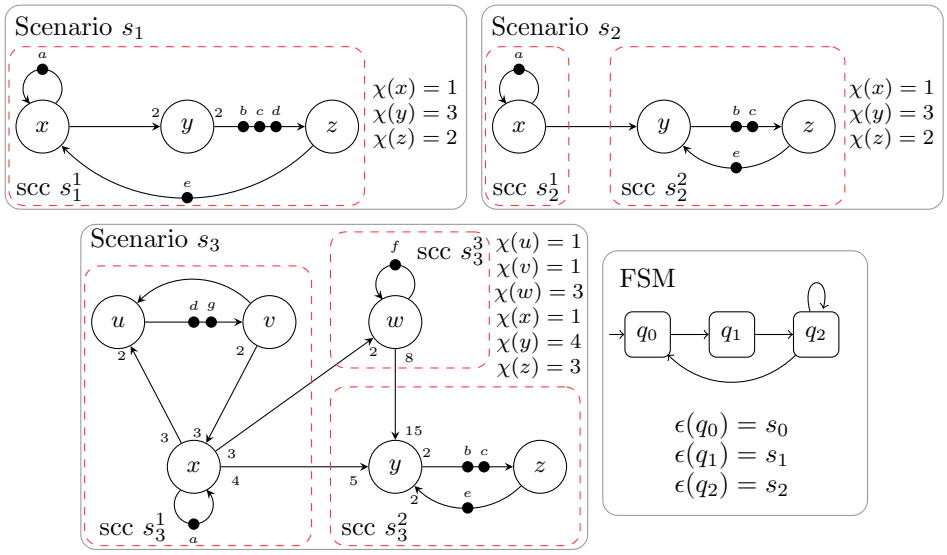


Figure 5.2: Example FSM-based SADF model

5.2.2 Inter-scenario Synchronization

The analysis techniques in [28,30] assume that the synchronization data between two scenarios are the *common initial tokens* they have. An initial token is common between two scenarios if it is in a *common channel*. A channel is common if it has the same source and destination actor in both scenarios. Synchronization through common initial tokens requires that 1) common channels must exist to enforce synchronization and 2) a common channel must have the same number of initial tokens in all scenarios. These may be restrictive requirements for real-life application graphs.

A simple example is the modeling of sub-frame detection in LTE (Figure 3.4), as discussed in Section 5.1. Synchronization is required to stall subsequent scenarios until the sub-frame type is detected. We cannot enforce this synchronization, since the mode controller actor (actor *mc* of scenario s_1) does not exist in other scenarios.

In this chapter, we use initial token labeling to model inter-scenario synchronization. Common initial tokens are explicitly defined by their identifier (label). Two initial tokens of two different scenarios are common if they have the same identifier. With this approach, common channels need not necessarily have equal number of initial tokens (cf. Section 5.5.1 for more discussion with an example).

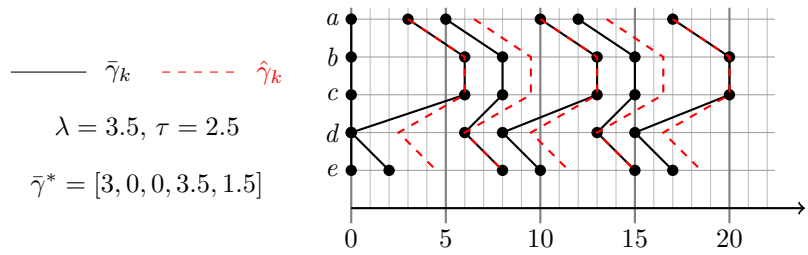


Figure 5.3: Time-stamp vectors of s_1 of Figure 5.2

5.2.3 Self-timed Boundedness

In compositional FSM-SADF analysis approaches, each scenario is first analysed individually. Then, performance guarantees are given for a sequence of scenarios compositionally (as also discussed in Section 5.5). To analyse scenarios individually, the analysis technique of [28, 30] assume scenarios are self-timed bounded. A self-timed bounded scenario requires bounded buffer sizes in a self-timed execution [34]. The time-stamp vector of a self-timed bounded scenario becomes periodic after a finite number of transient iterations. For instance, the matrix of scenario s_1 of Figure 5.2 is given by Equation (5.2). The time-stamp vectors of the first 5 iterations are shown by the black solid lines of Figure 5.3, computed using the relation $\bar{\gamma}_{k+1} = M \cdot \bar{\gamma}_k$, where $\bar{\gamma}_0 = \mathbf{u}[0]$ is a zero vector.

$$M = \begin{bmatrix} 2 & 3 & -\infty & -\infty & 2 \\ 5 & 6 & -\infty & -\infty & 5 \\ 5 & 6 & -\infty & -\infty & 5 \\ -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & 2 & -\infty & -\infty \end{bmatrix} \tag{5.2}$$

This graph becomes periodic after $\bar{\gamma}_2$. The shape of the vector (which is the normalized vector $\bar{\bar{\gamma}}_k$) repeats with *cyclicity* $\sigma = 2$ (i.e. every 2 iterations); for instance, $\bar{\gamma}_4 = \bar{\gamma}_2 + 7$. In the periodic phase, it completes 2 iterations every 7 time-units. Hence, the average duration of an iteration is $\lambda = 3.5$. λ is referred to as the *period*. The periodicity enables to simplify the analysis into an entirely periodic pattern. The idea is to find a *delay* $\tau \in \mathbb{R}$, and a vector $\bar{\gamma}^* \in \mathbb{R}^n_{\max}$, such that, for any $k > 0$,

$$\bar{\gamma}_k \preceq \hat{\bar{\gamma}}_k = \tau + \bar{\gamma}^* + \lambda k. \tag{5.3}$$

The vector $\hat{\gamma}_k$ is a *conservative linear upper-bound* to $\bar{\gamma}_k$, as shown by the red dashed lines in Figure 5.3. An example delay τ and vector $\bar{\gamma}^*$ for scenario s_0 are shown in Figure 5.3.

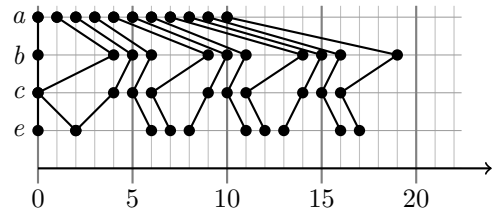


Figure 5.4: Time-stamp vectors of scenario s_2 of Figure 5.2

However, this periodic behavior does not hold for non-self-timed bounded scenarios. Consider scenario s_2 of Figure 5.2. The token production rate of channel (x, y) is faster than the consumption rate. Thus, the self-timed execution leads to an increasing accumulation of tokens in channel (x, y). Figure 5.4 shows the time-stamp vectors for the first 10 iterations of scenario s_1. Scenario s_2 has two strongly-connected components (SCCs), as shown in Figure 5.2. SCC s_2^1 runs at 1 iteration/time-unit, whereas SCC s_2^2 runs at 0.6 iteration/time-unit. This difference causes the time-stamps of tokens b, c, e to continuously diverge from token a. Thus, the different components of the scenario run at different rates of iterations. As a result, a recurrent vector does not exist and the algorithms of [28] cannot find parameters for linear upper-bound characterization.

Limiting FSM-SADF analysis only to self-timed bounded scenarios is restrictive. Even though strong-connectedness is a sufficient condition for self-timed boundedness [34], FSM-SADF scenarios of applications may become non-strongly connected (cf. LTE in Section 3.1 and WLAN in Section 3.2). These scenarios can potentially be non-self-timed bounded. Unlike SDF, a non-self-timed bounded scenario of a FSM-SADF is not necessarily a problem to unbounded channels. A channel with faster production rate in one scenario may be compensated by a faster consumption rate in another scenario, resulting in the net effect of a bounded channel.

One solution to the self-timed boundedness issue is converting scenarios to strongly connected graphs. In this case, however, the challenge becomes how to introduce extra channels and initial tokens, while guaranteeing the temporal behavior of the FSM-SADF is not compromised. Such extra channel additions may also result in pessimistic temporal bounds. In this chapter, we analyse FSM-SADF models without requiring every scenario to be strongly-connected or self-timed bounded. In the remainder of this chapter, we present the analysis in three steps. 1) Section 5.3 characterizes a scenario in terms of SCCs. 2) Section 5.4 discusses the analysis of a single scenario. 3) Section 5.5 generalizes the single scenario solution to a set of scenarios.

5.3 Condensation Graph

Each strongly-connected component (SCC) of a scenario is self-timed bounded. Hence, the conservative upper-bound characterization, discussed in Section 5.2.3, is applicable. Thus, each strongly-connected component c has a distinct period $\lambda_c \in \mathbb{R}_{\max}$. If each strongly-connected component of scenario s is contracted into a single node, a directed acyclic graph (DAG) is obtained. Each node of the DAG is then associated with a strongly-connected component.

We define the period of node n of the DAG as

$$\lambda_n \stackrel{\text{def}}{=} rep(c) \times \lambda_c \tag{5.4}$$

where c is the strongly-connected component of node n and $rep(c)$ is the *repetition* of c, which is defined as *the number of iterations of c within one iteration of scenario s*. Repetition is given as

$$rep(c) = \frac{\nu_s(a)}{\nu_c(a)} \tag{5.5}$$

where ν_s is the repetition vector of scenario s, ν_c is the repetition vector of c and a is any actor of c. For instance, the repetition vector of scenario s_3 of Figure 5.2 is given as $\nu = \{(u, 15), (v, 15), (w, 15), (x, 10), (y, 8), (z, 16)\}$. The repetition vector of strongly-connected component s_3^1 is $\nu_c = \{(u, 3), (v, 3), (x, 2)\}$. Hence, s_3^1 has a repetition of 5.

We next partition a scenario into a set of maximal *self-timed bounded components (STBCs)*. We achieve this through repetitive SCC grouping. A STBC is not necessarily strongly-connected but it is self-timed bounded. Consider an edge (m, n) of the DAG. The maximum rate at which the strongly-connected component of node n completes iterations of the scenario is $\frac{1}{\lambda_n}$ iterations per time-unit. However, if $\lambda_m \geq \lambda_n$, the SCC of n cannot run faster than the SCC of m in the execution of the scenario, as it consumes data produced by the SCC of m. As a result, node n runs at the same rate as node m. Combining the strongly-connected components of these two nodes, as per their original connections in the scenario, gives a STBC c_{mn}. We group the two nodes m and n into one node mn. Node mn is now associated with the STBC c_{mn}. The period of node mn is the slower (longer) of the the two strongly-connected components, hence $\lambda_{mn} = \lambda_m$.

Repetitive grouping of nodes in such a manner terminates when $\lambda_m < \lambda_n$ for any edge (m, n) of the DAG. This operation partitions the scenario into a set of STBCs. After such node grouping, some channels of the scenario will be left that are not part of any STBC. These channels only connect the different STBCs. We refer to such channels as *feed-forward channels*. The feed-forward channels are basically the unbounded channels.

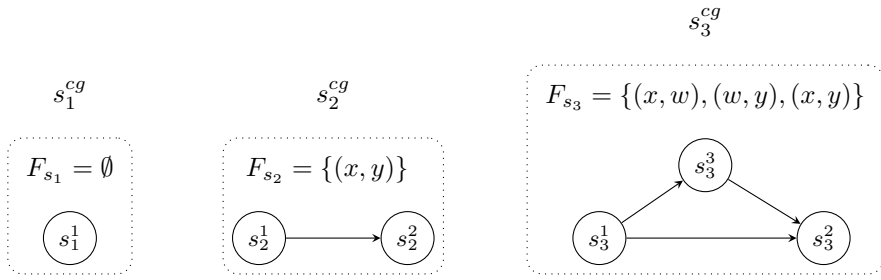


Figure 5.5: Condensation graphs of Figure 5.2

The resulting DAG, after the repetitive node merging, is referred to as *condensation graph*, as defined in Definition 12.

Definition 12 (Condensation Graph)**.** *Let BC_s and F_s are the set of STBCs and feed-forward channels of scenario s, respectively. The condensation graph s^{cg} of s is a directed graph (V, E) of a set V of vertices and a set $E \subseteq V \times V$ of edges. Each feed-forward channel $c_f \in F_s$ has exactly one edge $e = (v_s, v_d) \in E$ assigned to it such that the source v_s and destination v_d vertices are respectively labeled with the source and destination STBCs of c_f.*

The feed-forward channels and the condensation graphs of the scenarios of Figure 5.2 are shown in Figure 5.5. F_{s_1}, F_{s_2} and F_{s_3} are respectively the set of feed-forward channels of scenario s_1, s_2 and s_3, which also have 1, 2 and 3 STBCs. (The STBCs are shown in red dashed boxes in Figure 5.2). The label inside each vertex of Figure 5.5 indicates the STBC, which the vertex corresponds to.

We say initial token t of scenario s belongs to vertex v_t of condensation graph s^{cg} if t is either in the self-time bounded component of vertex v_t or in the feed-forward channel of an outgoing edge of vertex v_t. We use the *levels of vertices* to determine dependencies between initial tokens. The level of a vertex is defined inductively as follows: for a root vertex (which has no incoming edges), $level(v) = 0$ and for any other vertex, $level(v) = 1 + \max_{v \in v^-} level(v)$, where v^- is the set of all direct predecessors of v.

If $level(v_t) < level(v_u)$ then initial token t has no dependency from initial token u. Hence, the $(max, +)$ matrix of the scenario has $[M]_{\zeta(t)\zeta(u)} = -\infty$, as discussed in Section 2.4. The levels of vertices are also used to construct a *topological order* of the condensation graph, as defined by Definition 13.

Definition 13 (Topological Order)**.** *A* topological order *of s^{cg} is a linear ordering of its vertices $\langle v^1, v^2, \cdots, v^n \rangle$, such that v^i comes before v^j only if $level(v^i) \leq level(v^j)$, where for $i \in \mathbb{N}$, v^i denotes the i^{th} vertex in the order.*

$$\begin{bmatrix} t'_a \\ t'_b \\ t'_c \\ t'_d \end{bmatrix} = \begin{bmatrix} 1 & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 3 \\ -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & 2 & -\infty \end{bmatrix} \cdot \begin{bmatrix} t_a \\ t_b \\ t_c \\ t_d \end{bmatrix}$$

Figure 5.6: Matrix partitioning of scenario s_2 of Figure 5.2

We use topological order of vertices to present the matrix of a scenario in a lower triangular form. Shuffle time-stamp vector $\bar{\gamma}_k$ and matrix M based on a given topological order: I.e. for any two initial tokens t and u in two different STBCs, assign $\zeta(t) < \zeta(u)$ only if $level(v_t) \leq level(v_u)$. As a result, the indices of those initial tokens that belong to the same vertex becomes consecutive.

Hence, we can partition $\bar{\gamma}_k$ into a set of n disjoint sub-vectors $\langle \bar{\gamma}_k^1, \bar{\gamma}_k^2, \cdots, \bar{\gamma}_k^n \rangle$, where $\bar{\gamma}_k^i$ denotes the sub-vector of vertex v^i. Similarly, matrix M is also partitioned into $n \times n$ sub-matrices. The resulting matrix has a triangular form, as shown in the recurrence relation of Equation (5.6).

$$\begin{bmatrix} \bar{\gamma}_{k+1}^1 \\ \bar{\gamma}_{k+1}^2 \\ \vdots \\ \bar{\gamma}_{k+1}^n \end{bmatrix} = \begin{bmatrix} M^{11} & -\boldsymbol{\infty} & \cdots & -\boldsymbol{\infty} \\ M^{21} & M^{22} & \cdots & -\boldsymbol{\infty} \\ \vdots & \vdots & \ddots & \vdots \\ M^{n1} & M^{n2} & \cdots & M^{nn} \end{bmatrix} \cdot \begin{bmatrix} \bar{\gamma}_k^1 \\ \bar{\gamma}_k^2 \\ \vdots \\ \bar{\gamma}_k^n \end{bmatrix} \tag{5.6}$$

The dimension of sub-matrix M^{ij} is $m \times n$ where $m = |\bar{\gamma}_k^i|$ and $n = |\bar{\gamma}_k^j|$. In Equation (5.6), $-\boldsymbol{\infty}$ represents a sub-matrix whose entries are all $-\infty$. Sub-matrices on the diagonal, i.e. M^{ii}, can be computed from the matrices of the STBCs of the scenario. Given matrix M_{c_i} of the self-timed bounded component c_i of vertex v^i, M^{ii} is computed by multiplying M_{c_i} with itself $rep(c_i)$ times so that it accounts for a complete iteration of the scenario. The partitioning of scenario s_2 of Figure 5.2, whose matrix discussed in Section 2.4, is shown in Figure (5.6).

Such kind of matrix partitioning is possible for any reducible $(max, +)$ matrix [43]. The correlation with self-timed bounded components of a scenario is as follows. Any sub-vector $\bar{\gamma}_{k+1}^i$ is related to other sub-vectors as given by Equation (5.7), derived directly from Equation (5.6).

$$\begin{aligned} \bar{\gamma}_{k+1}^i &= \max\left(M^{i1} \cdot \bar{\gamma}_k^1,\ M^{i2} \cdot \bar{\gamma}_k^2,\ \cdots,\ M^{in} \cdot \bar{\gamma}_k^n\right) \\ &= \max\left(\max_{j=1}^{i-1}\left(M^{ij} \cdot \bar{\gamma}_k^j\right),\ M^{ii} \cdot \bar{\gamma}_k^i\right) \end{aligned} \tag{5.7}$$

The term $\max_{j=1}^{i-1}\left(M^{ij}\cdot\bar{\gamma}_k^j\right)$ of Equation (5.7) represents the data dependencies of vertex v^i with all its predecessors in the topological order. Equation (5.7) is a valuable relation to explain a periodic behavior in non-self-timed bounded scenarios, discussed next in Section 5.4.

5.4 Analyzing a Single Scenario

In this section, we derive conservative upper-bounds to iterations of a scenario ($\hat{\bar{\gamma}}_k$), as discussed in Section 5.2.3. We present a generalized approach that allows analysing any arbitrary scenario, without requiring self-timed boundedness. A valuable concept in our approach is the *periodicity* of a (non-strongly connected) scenario in a self-timed execution.

The self-timed execution of a SDFG $g = (A, C, I, \chi, \rho, \iota)$ enters a periodic phase after a finite number of transient iterations. In the periodic phase, the timing distance between consecutive time-stamp vectors $\bar{\delta}_{k+1} = \bar{\gamma}_{k+1} - \bar{\gamma}_k$ becomes periodic; I.e. $\bar{\delta}_{k+1}$ repeats every $\sigma \in \mathbb{N}$ iterations. This periodic behavior is intuitively explained as follows.

Any vertex of the condensation graph of g is slower than any of its predecessors. This implies that in a self-timed execution of the scenario, tokens increasingly accumulate in all feed-forward channels. Such token accumulations decouple the executions of neighboring vertices. Consequently, the STBC of each vertex eventually executes independently, after a finite number of transient iterations.

When a vertex v_i enters the independent execution phase, it has no dependencies with its predecessor vertices. Thus, Equation (5.7) becomes $\bar{\gamma}_{k+1}^i = M^{ii}\cdot\bar{\gamma}_k^i$. Eventually, the entire $\bar{\gamma}_k$ attains periodicity, as given by Equation 5.8.

$$\forall k \geq p, \quad \bar{\gamma}_{k+\sigma} = \bar{\gamma}_k + \sigma\eta. \tag{5.8}$$

where $p \in \mathbb{N}$ is the first iteration of the periodic phase, $\sigma \in \mathbb{N}$ is the LCM of the cyclicities of the STBCs and vector $\eta \in \mathbb{R}_{\max}^n$ is the collection of the periods of the vertices of the condensation graph, as given by Equation (5.9), where v_i is the vertex of initial token $i \in I$.

$$[\eta]_{\zeta(i)} = \lambda_{v_i} \tag{5.9}$$

The vector η is called *cycle-time vector* in $(max, +)$ algebra [43]. It indicates the growth rate of a vector in a recurrence relation of the form of Equation 2.1. The cycle-time vector of a regular matrix is unique and finite [43]. The maximum entry of η, $\|\eta\|$ is the period of the slowest vertex. Hence, the worst-case throughput of a scenario is given as $\frac{1}{\|\eta\|}$ iterations/time-unit. For instance, scenario s_1 of

Figure 5.2 has $\eta = [1, 1.667, 1.667, 1.667]$. The maximum entry $\|\eta\| = 1.667$ is the period of vertex s_1^1, which is the slowest of the two vertices of scenario s_1.

As in the case of self-timed bounded scenarios (Section 5.2.3), a conservative upper-bound $\hat{\bar{\gamma}}_k$ can also be computed for non-self-timed bounded scenarios, as given by Equation (5.10). With such conservative characterization, we can derive a bound to the finishing time of any iteration $k > 0$.

$$\bar{\gamma}_k \preceq \hat{\bar{\gamma}}_k = \mathbf{u}[\tau] + \bar{\gamma}^* + k\eta \tag{5.10}$$

We refer to $\tau \in \mathbb{R}$ as *delay* and $\bar{\gamma}^* \in \mathbb{R}^n_{\max}$ as *initial vector*. To compute these two parameters, we use the concept of *generalized eigenmode* from $(max, +)$ algebra, defined in Definition 14 [43]. Generalized eigenmode is a generalization of eigenvectors to reducible $(max, +)$ matrices, which are matrices of non-strongly connected graphs.

Definition 14. (Generalized Eigenmode) *Given the cycle-time vector η of matrix M, a generalized eigenmode of M is a pair $(\eta, \nu) \in \mathbb{R}^n_{\max} \times \mathbb{R}^n_{\max}$ such that for any $p \in \mathbb{N}^0$, $M(p\eta + \nu) = (p + 1)\eta + \nu$.*

A generalized eigenmode exits for any *regular* matrix [43]. A $(max, +)$ matrix M is regular if it has no row whose elements are all $-\infty$. A scenario cannot have an initial token that has no dependency with any other initial tokens (including itself) in the graph, as discussed in Section 2.4. Thus, the matrix of any scenario is regular.

The generalized eigenmode implies that if we execute the recurrence relation $\bar{\gamma}'_k = M\bar{\gamma}'_{k-1}$ starting from $\bar{\gamma}'_0 = \nu$, the vectors advance exactly by η, i.e. $\bar{\gamma}'_{k+1} - \bar{\gamma}'_k = \eta$. This makes ν a suitable candidate for initial vector $\bar{\gamma}^*$, because the average difference between consecutive iterations in the periodic phase is also equal to η (cf. Equation (5.8)). Thus, these vectors $(\bar{\gamma}'_k)$ can be upper-bounds to $\bar{\gamma}_k$ if shifted sufficiently. Delay τ is such a shifting distance, i.e. $\bar{\gamma}_k \preceq \tau + \bar{\gamma}'_k$. A tight delay, however, is key to avoid overly pessimistic analysis.

The generalized eigenvector of a matrix is not unique. Lemma 1 presents one approach for computing a generalized eigenvector using any σ consecutive iterations in the periodic phase, generalizing the method of [28].

Lemma 1. (Generalized Eigenvector) *Let $\bar{\gamma}_k$ be a time-stamp vector in the periodic phase where $1 \leq k \leq \sigma$. I.e. $\bar{\gamma}_{\sigma+1} = \bar{\gamma}_1 + \sigma\eta$. Then, the evaluation $\nu = \max_{1 \leq k \leq \sigma}(\bar{\gamma}_k - k\eta + \sigma\eta)$ is a generalized eigenvector.*

Proof.

$$\begin{aligned} x &\overset{\text{def}}{=} M(p\eta+\nu) \\ &= M(p\eta+\max_{1\le k\le\sigma}(\bar{\gamma}_k-k\eta+\sigma\eta)) && \text{by substitution of } \nu \\ &= \max_{1\le k\le\sigma} M(\bar{\gamma}_k-k\eta+\sigma\eta+p\eta) && \text{by } (max,+) \text{ multiplication linearity} \\ &= \max_{1\le k\le\sigma} M\bar{\gamma}_k-k\eta+\sigma\eta+p\eta \end{aligned}$$

The last step is because, in the periodic phase, $\max_{j=1}^{i-1}[M]_{ij}\cdot\bar{\gamma}_k^j \preceq [M]_{ii}\cdot\bar{\gamma}_k^i$, and if $i<j$, then $c[\eta]_i \le c[\eta]_j$ for any $c\in\mathbb{N}^0$.

We then get

$$\begin{aligned} &= \max_{1\le k\le\sigma}\bar{\gamma}_{k+1}-k\eta+\sigma\eta+p\eta && \text{using recurrence relation of Equation (2.1)} \\ &= \max_{1\le k\le\sigma}\bar{\gamma}_{k+1}-k\eta+\sigma\eta+p\eta+\eta-\eta && \text{adding } \eta-\eta \\ &= (p+1)\eta+\max_{1\le k\le\sigma}\bar{\gamma}_{k+1}-(k+1)\eta+\sigma\eta && \text{after algebraic steps} \\ &= (p+1)\eta+\max_{1\le k\le\sigma}\bar{\gamma}_k-k\eta+\sigma\eta && \text{by periodic property } \bar{\gamma}_{\sigma+1}=\bar{\gamma}_1+\sigma\eta \end{aligned}$$

The last step then gives $x=(p+1)\eta+\nu$. Hence, (η,ν) is a generalized eigenmode by Definition 14. □

Algorithm 3 Compute delay τ

```
1: ComputeDelay(η, γ̄_0, γ̄*)
2: k := 0, τ := −∞, γ̃̄ := ∅,
3: (γ̄_k, δ̄_k) := (γ̄_0, u[−∞])
4: loop
5:    γ̃̄ := γ̃̄ ∪ {(γ̄_k, δ̄_k)}, k := k + 1
6:    γ̄_k := M · γ̄_{k−1}
7:    δ̄_k := γ̄_k − γ̄_{k−1}
8:    τ := max(τ, ‖γ̄_k − γ̄* − kη‖)
9:    if ∃(γ̄_m, δ̄_m) ∈ γ̃̄ : δ̄_k = δ̄_m then
10:      if γ̄_k − γ̄_m = (k − m) · η then
11:         return τ
12:      end if
13:   end if
14: end loop
```

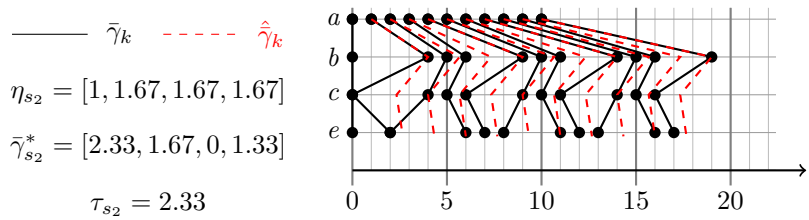


Figure 5.7: Time-stamp vectors of scenario s_2 of Figure 5.2

An algorithm for computing a delay τ from a given $\bar{\gamma}^*$ and η is given in Algorithm 3, which generalizes the delay computation algorithm of [28]. Algorithm 3 finds a delay $\tau \in \mathbb{R}$ that satisfies Equation (5.11) that is derived from Equation (5.10).

$$\mathbf{u}[\tau] \succeq \bar{\gamma}_k - \bar{\gamma}^* - k\eta \tag{5.11}$$

In the algorithm, $\widetilde{\bar{\gamma}}$ (line 2) is the set of pairs $(\bar{\gamma}_k, \bar{\delta}_k)$ of a time-stamp vector and its difference with the previous time-stamp vector. All such pairs are added to the set (line 5) to test if the execution has reached a periodic phase. The execution begins with a zero vector $\bar{\gamma}_0$ (line 3). The algorithm executes the recurrence relation (line 6) of Equation (2.1) until the periodic phase is detected (lines 9-10). It tracks the difference vector (line 7) to detect the periodic phase. After the periodic phase is reached, τ does not change any more and therefore, the algorithm stops.

Figure 5.7 shows the conservative upper-bounds ($\hat{\bar{\gamma}}_k$) of the time-stamp vectors, shown in Figure 5.4 (scenario s_2 of Figure 5.2). The initial vector $\bar{\gamma}^*_{s_1}$ and delay τ_{s_1} are computed by Lemma 1 and Algorithm 3, respectively. Using such conservative upper-bounds for every individual scenario, we can reason about completion times of iterations irrespective of the scenario sequence they are executed in, as discussed next in Section 5.5.

5.5 FSM-SADF Throughput Analysis

In the execution of a scenario sequence, each scenario is executed for one iteration. The end of the k^{th} iteration is marked by time-stamp vector $\bar{\gamma}_k$, which is used as a starting vector for the $(k+1)^{th}$ iteration, possibly in a different scenario. This means, $\bar{\gamma}_{k+1} = M_s \cdot \bar{\gamma}_k$ where M_s is the $(max, +)$ matrix of the scenario at the $(k+1)^{th}$ iteration. In this section, we analyse the WCT of FSM-SADF, as given by Definition 11. Section 5.5.1 defines time-stamp vector of FSM-SADF. Section 5.5.2 and 5.5.3 present conservative and exact analysis techniques, respectively.

5.5.1 Time-stamp Vector of an FSM-SADF

The scenarios of an FSM-SADF may have different sets of initial tokens. If so, their respective matrices may have inconsistent dimensions for the recurrence relation, $\bar{\gamma}_{k+1} = M_s \cdot \bar{\gamma}_k$. Due to this reason, the matrix of a scenario needs to be extended to accommodate those initial tokens that exist in other scenarios but not in the scenario.

Denote the set of initial tokens of scenario s as I_s. Denote the set of initial tokens of FSM-SADF (S, f) as

$$I_{\mathcal{F}} = \bigcup_{s \in S} I_s.$$

Similar to Definition 6, we associate indices to initial tokens; $\zeta_{\mathcal{F}} : I_{\mathcal{F}} \leftrightarrow \underline{n}$, where $n = |I_{\mathcal{F}}|$. A time-stamp vector of (S, f) is then a vector $\bar{\gamma}_k \in \mathbb{R}^n_{\max}$ that has exactly one row for every initial token $i \in I_{\mathcal{F}}$ at index $\zeta_{\mathcal{F}}(i)$.

A *super-matrix* of scenario s, $M_s^{\mathcal{F}} \in \mathbb{R}^{n \times n}_{\max}$, is constructed from the original $(max, +)$ matrix of s, M_s, as given by Equation (5.12). $M_s^{\mathcal{F}}$ accommodates every initial token $i \in I_{\mathcal{F}}$. When scenario s is executed at some iteration k, the super-matrix $M_s^{\mathcal{F}}$ enables to preserve the time-stamps of every initial token $i \in I_{\mathcal{F}} \backslash I_s$ ($A \backslash B$ denotes set A minus set B) for subsequent iterations beyond k.

$$[M_s^{\mathcal{F}}]_{\zeta_{\mathcal{F}}(i)\zeta_{\mathcal{F}}(j)} = \begin{cases} [M_s]_{\zeta_s(i)\zeta_s(j)} & i, j \in I_s \\ 0 & i = j \wedge i, j \in I_{\mathcal{F}} \backslash I_s \\ -\infty & i \neq j \wedge i \in I_{\mathcal{F}} \backslash I_s \vee j \in I_{\mathcal{F}} \backslash I_s \end{cases} \tag{5.12}$$

E.g. the initial tokens of the scenarios of Figure 5.2 are $I_{s_0} = \{a, b, c, d, e\}$, $I_{s_1} = \{a, b, c, e\}$ and $I_{s_2} = \{a, b, c, d, e, f, g\}$. Therefore, $I_{\mathcal{F}} = \{a, b, c, d, e, f, g\}$. Extending the matrix of scenario s_1 (Equation 2.2) to accommodate initial tokens $I_{\mathcal{F}} \backslash I_{s_1} = \{d, f, g\}$ yields matrix $M_{s_1}^{\mathcal{F}}$ of Equation (5.13). The indices of initial tokens of $M_{s_1}^{\mathcal{F}}$ are $\zeta_{\mathcal{F}} = \{(a, 1), (b, 2), (c, 3), (d, 4), (e, 5), (f, 6), (g, 7)\}$. Executing scenario s_1 (i.e $\bar{\gamma}_{k+1} = M_{s_1}^{\mathcal{F}} \cdot \bar{\gamma}_k$) preserves the time-stamps of tokens $\{d, f, g\}$, as $[\bar{\gamma}_k]_i = [\bar{\gamma}_{k+1}]_i$ for $i \in \{4, 6, 7\}$.

$$M_{s_1}^{\mathcal{F}} = \begin{bmatrix} 1 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & -\infty & 3 & -\infty & -\infty \\ -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & 2 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} \tag{5.13}$$

5.5.2 Conservative Worst-Case Throughput

This section presents three different conservative WCT analysis techniques. The WCT of FSM-SADF is the minimum among the WCTs of the possible scenario sequences, as given by Definition 11. The WCT of a sequence is defined as the minimum long-run average of completed iterations per time-unit. The FSM can potentially specify infinitely many scenario sequences. However, all sequences are basically formed by concatenation of the cycles of the FSM. Hence, a conservative WCT of FSM-SADF can be computed by a maximum cycle mean (MCM) analysis on the FSM.

For this purpose, each FSM state $q \in Q$ is assigned a weight $w_{\epsilon(q)} \in \mathbb{N}$. A valid weight is a conservative bound to the duration of one iteration of the scenario of q. This narrows down the WCT analysis problem to finding such conservative bounds of scenarios.

Approach 1: Scenario Graph

One simple approach is to abstract from scenario transitions; i.e to derive a bound for one iteration of a scenario, regardless of other scenarios in the FSM-SADF. Evaluating Equation (5.10) for the first iteration ($k = 1$) gives

$$\bar{\gamma}_1 = M_s^F \bar{\gamma}_0 \preceq \mathbf{u}[\tau_s] + \bar{\gamma}_s^* + \eta_s \tag{5.14}$$

where $\bar{\gamma}_0 = \mathbf{u}[0]$ and η_s, $\bar{\gamma}_s^*$ and τ_s are the cycle-time vector, the generalized eigenvector and delay of scenario $s \in S$, computed by Equation 5.9, Lemma 1 and Algorithm 3, respectively.

Let scenario s is executed at iteration k of a scenario sequence specified by the FSM. Generally, $\bar{\bar{\gamma}}_{k-1} \preceq \bar{\gamma}_0$ since all entries of the normalized vector $\bar{\bar{\gamma}}_{k-1}$ are less than or equal to 0. Then, we get

$$\begin{aligned} M_s^F \bar{\bar{\gamma}}_{k-1} &\preceq M_s^F \bar{\gamma}_0 \text{.....from monotonicity} \\ M_s^F \bar{\bar{\gamma}}_{k-1} &\preceq \mathbf{u}[\tau_s] + \bar{\gamma}_s^* + \eta_s \text{.....from Equation (5.14)} \\ \|M_s^F \bar{\bar{\gamma}}_{k-1}\| &\leq \|\mathbf{u}[\tau_s] + \bar{\gamma}_s^* + \eta_s\| \end{aligned} \tag{5.15}$$

The duration of iteration k (i.e. the distance between the finishing times of iteration k and $k-1$) is bounded by $\|M_s^F \bar{\bar{\gamma}}_{k-1}\|$. As a result, the bound to the first iteration of scenario s, given by Equation (5.16), is a bound to an iteration of scenario s at any iteration k, in any scenario sequence (since we have no assumptions regarding the scenario of iteration $k-1$).

$$w_s = \|\mathbf{u}[\tau_s] + \bar{\gamma}_s^* + \eta_s\| \tag{5.16}$$

The inverse of the maximum among all such weights is a valid WCT of FSM-SADF. We refer to this technique as the *scenario graph* approach and the WCT ρ_{sg} is given by Equation (5.17) .

$$\rho_{sg} = \frac{1}{\max_{s \in S} w_s}. \tag{5.17}$$

Approach 2: Global Initial Vector

A more accurate analysis is obtained if conservative bounds of scenarios are computed, which are tailored to the scenarios in the FSM-SADF. Define a global initial vector

$$\bar{\gamma}_S^* = \max_{s \in S} \bar{\gamma}_s^* \tag{5.18}$$

where $\bar{\gamma}_s^*$ is the generalized eigenvector of scenario $s \in S$, computed by Lemma 1. For each scenario s, compute delay τ_s (Algorithm 3) using the cycle-time vector of the scenario η_s and $\bar{\gamma}_S^*$. Following a similar argument as in Approach 1,

$$w_s = \|\mathbf{u}[\tau_s] + \bar{\gamma}_S^* + \eta_s\| \tag{5.19}$$

is a bound to a single iteration of scenario s at any iteration k, in any scenario sequence. The MCM of the FSM gives a valid WCT ρ_{gv}, referred to as *global initial vector* approach, as given by Equation (5.20). C denotes the set of cycles of the FSM. A cycle $c \in C$ is a sequence of states $\langle q_{i+0}, q_{i+1}, \cdots, q_{i+n}, q_{i+0} \rangle$, such that there exists a transition between consecutive states and no state appears more than once.

$$\rho_{gv} = \frac{1}{\max\limits_{c \in C} \frac{1}{|c|} \sum\limits_{\forall q \in c} w_{\epsilon(q)}} \tag{5.20}$$

Approach 3: Local Initial Vector

The global initial vector approach can also be further refined by computing local vectors, shown in Equation (5.21) between every FSM transition $(p, q) \in T$, where $r = \epsilon(p)$ and $s = \epsilon(q)$.

$$\bar{\gamma}_{rs}^* = \max(\bar{\gamma}_r^*, \bar{\gamma}_s^*) \tag{5.21}$$

For FSM transition (p, q), compute delay τ_{rs} (Algorithm 3) using cycle-time vector η_s and $\bar{\gamma}_{rs}^*$. Hence, w_{rs}, given by Equation (5.22), is a bound to a single iteration of scenario s in any scenario transition r to s, in any scenario sequence.

$$w_{rs} = \|\mathbf{u}[\tau_{rs}] + \bar{\gamma}^{*}_{rs} + \eta_s\| \tag{5.22}$$

The MCM of the FSM gives a valid WCT referred to as *local initial vector* approach, as given by Equation (5.23), where p is the preceding FSM state of q in cycle c.

$$\rho_{lv} = \frac{1}{\max\limits_{c \in C} \frac{1}{|c|} \sum\limits_{\forall q \in c} w_{\epsilon(p)\epsilon(q)}} \tag{5.23}$$

5.5.3 Exact Worst-Case Throughput

Two exact WCT computation techniques for FSM-SADF, whose scenarios are all self-timed bounded, are presented in [30]. In this section, we show that one of these techniques is applicable to compute the exact WCT of FSM-SADF whose scenarios are possibly non-self-timed bounded. This technique is known by the name $(max, +)$ *automaton* [26]. It is based on *timed-event graphs* of the scenarios of the FSM-SADF.

A timed-event graph is a weighted directed graph representation of a $(max, +)$ matrix. It depicts the timing distance between initial tokens. It is also known as *communication graph* in $(max, +)$ literature [43]. The timed-event graph of an $n \times n$ matrix has n nodes. The graph has an edge from node i to node j if $[M]_{ij} \neq -\infty$ and the weight of the edge equals $[M]_{ij}$. The timed-event graph of Equation (2.2) is shown in Figure 5.8.

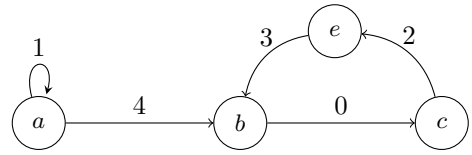


Figure 5.8: Timed-event graph of Equation (2.2)

According to the $(max, +)$ automaton technique, the worst-case throughput of an FSM-SADF is equal to the MCM of the *throughput graph* of the FSM-SADF. The throughput graph $\mathbf{G} = (V, E)$ is a weighted directed graph that is constructed from the timed-event graphs of the scenarios and the FSM of the FSM-SADF.

First, for each FSM state $q \in Q$, place the timed-event graph of its scenario, $\epsilon(q)$. Then, for every FSM transition $(p, q) \in T$, add edge from node i of the timed-event graph of $\epsilon(p)$ to node j of the timed-event graph of $\epsilon(q)$ if $[M^{F}_{\epsilon(q)}]_{\zeta_F(j)\zeta_F(i)} \neq -\infty$.

A weight $w(e)$, shown in Equation (5.24), is then set for each edge $e = (i, j)$ as follows.

$$w(e) = [M^{\Gamma}_{\epsilon(q)}]_{\zeta_\Gamma(j)\zeta_\Gamma(i)}. \tag{5.24}$$

The weight of an edge is the minimum timing distance between two initial tokens in consecutive iterations. Hence, it gives the length of an iteration relative to these two dependent initial tokens. For infinitely long execution of the FSM-SADF (as in streaming applications), all such distances reside within the cycles of the throughput graph. Therefore, the inverse of the maximum average weight among all cycles gives the exact WCT as defined in Definition 11.

The exact WCT of FSM-SADF is then given by the MCM of the throughput graph as given in Equation 5.25, where C denotes the set of all cycles in the throughput graph and E^c denotes the set of edges in cycle c.

$$\frac{1}{\rho_{ma}} = \max_{c \in C} \frac{1}{|E^c|} \sum_{\forall e \in E^c} w(e) \tag{5.25}$$

Unlike the conservative approaches of Section 5.5.2, Equation 5.25 gives the exact WCT that may occur in the execution of the FSM-SADF. The improvement in accuracy, however, comes at a price of run-time, as discussed next in Section 5.6.

5.6 Evaluation

This section demonstrates the applicability of the presented throughput analysis techniques and evaluates their scalability. Section 5.6.1 assesses the relative conservativeness of the analysis techniques with different dataflow graphs. Section 5.6.2 evaluates the scalability of the analysis techniques as the number of initial tokens and scenario graphs increase in FSM-SADF models.

5.6.1 Conservativeness

Table 5.1 shows the worst-case throughputs (WCTs) of different FSM-SADF graphs, according to the different analysis techniques presented in Section 5.5. The second and third columns of the table indicate the number of initial tokens (#tokens) and number of scenarios (#scenarios) of the FSM-SADF graphs. Columns ρ_{sg}, ρ_{gv} and ρ_{lv} are conservative WCTs, according to Equations (5.17), (5.20) and (5.23), respectively. Column ρ_{ma} is the exact WCT, according to Equation (5.25). The $(max, +)$ automaton ρ_{ma} gives the exact WCT of the FSM-SADF graphs. Therefore, it is used as a reference to compare the relative pessimism (accuracy) of the other three techniques.

Table 5.1: WCT($\times 10^{-4}$ iterations/time-unit)

Appl.	**Properties**		**Throughput**				**Run-time**
	# tokens	# scenarios	ρ_{sg}	ρ_{gv}	ρ_{lv}	ρ_{ma}	**(msec)**
Figure 5.2	9	3	1.6e2	1.7e2	1.7e2	1.7e2	4/0
WLAN	9	4	1.7	2.1	2.5	2.5	4/0
LTE	17	5	4.8	8.1	9.9	14.5	12/4
Satellite	22	1	7.6	9.4	9.4	9.4	124/64
MP3	27	8	1.0e-3	2.4e-3	2.4e-3	3.1e-3	48/32
RVC-MPEG	255	16	0.1	0.1	0.16	0.31	720/1076

Table 5.1 shows that ρ_{lv} gives a better result than the other two, and ρ_{gv} is better than ρ_{sg}. This observation also holds in general. ρ_{sg} does not consider the pipelined or overlapped execution of scenarios. Hence, it is the least accurate among the three approaches. ρ_{lv} gives a better result than ρ_{gv} as it performs a refined analysis using local transition delays. However, such analysis improvements come at a price of additional analysis run-times, as discussed in Section 5.6.2.

The last column in the table shows the run-times of the analysis techniques. The entries (*conservative/exact*) of the column shows the maximum run-time among the three conservative techniques and the run-time of the exact technique, respectively. For the RVC-MPEG application, for example, the maximum analysis run-time with the conservative techniques is $720msec$, while the exact $(max, +)$ automaton approach takes $1076msec$. The run-times of the analysed applications in Table 5.1 are quite low (almost below $1sec$). However, the table shows that the analysis run-time increases as the number of initial tokens (#tokens) and scenarios (#scenarios) increase. The next section evaluates the scalability of the throughput analysis techniques as these parameters increase further.

5.6.2 Scalability

The two main factors that affect the run-times of the analysis techniques are *the number of FSM states* and *the number of initial tokens*.

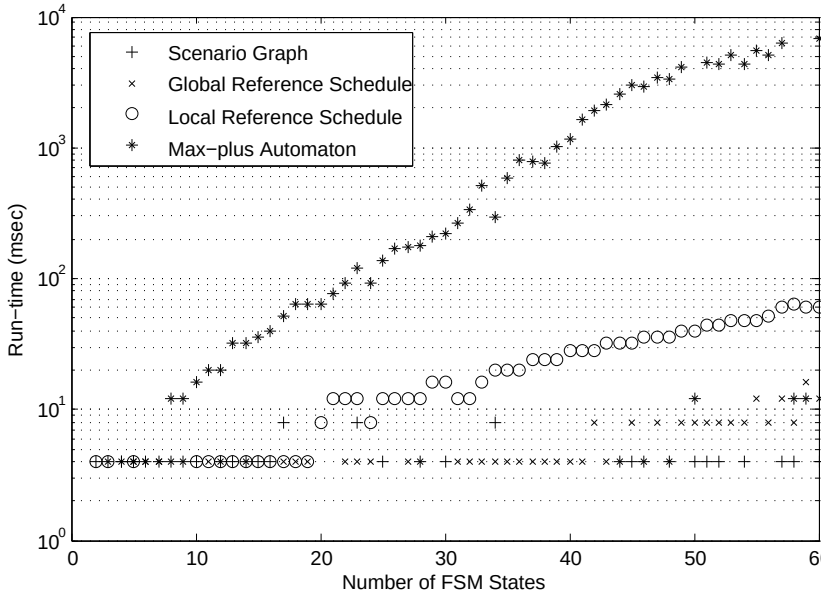


Figure 5.9: Analysis run-time as the number of FSM states increases

Number of FSM states

If a dynamic application is modeled in detail, the number of scenarios may substantially increase. As a result, the size of the FSM may also increase. Conservative techniques perform MCM analysis on the FSM, while the $(max, +)$ automaton approach constructs the throughput graph (cf. Section 5.5.3) by replacing each state of the FSM with the timed-event graph (cf. Section 5.5.3) of the scenario of the state. Figure 5.9 shows how the run-times of the different analysis techniques scale with increasing number of FSM states. The evaluation is conducted by generating a random synthetic FSM-SADF graph using the SDF3 [4] dataflow tool. The generated graph has five scenarios and 22 initial tokens. The number of FSM states of the graph is then varied from 2 to 60. The figure shows that conservative techniques in general scale better than the exact technique. The conservative techniques perform a MCM analysis on the FSM, which has an order of complexity $\mathcal{O}(Q^3)$, since the FSM has $|Q|$ states and at most $|Q|^2$ edges. The exact technique, on the other hand, performs a MCM analysis on the throughput graph, which has an order of complexity $\mathcal{O}(V^3)$, where V is the number of vertices of the throughput graph. $V = |Q| \times |I_F|$, where I_F is the number of initial initial tokens of the FSM-SADF.

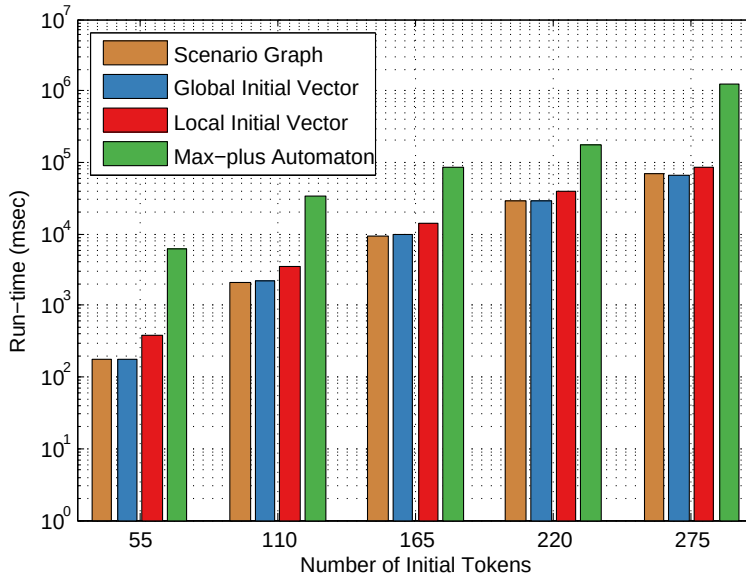


Figure 5.10: Analysis run-time vs. initial tokens (#FSM states=50)

Number of initial tokens

The above discussion also shows that the run-time of the $(max, +)$ automaton technique has time-complexity with the number of initial tokens. This is demonstrated in Figure 5.10, which is constructed by increasing the number of initial tokens on a generated FSM-SADF graph, which has 50 FSM states. The figure also shows the scalability of the conservative techniques. The dimension of the $(max, +)$ matrix of a scenario is $n \times n$, where n is the number of initial tokens. Hence, the computational workload of such matrices is expected to grow with the number of initial tokens. In particular, the number of initial tokens in FSM-SADF may increase significantly when additional dataflow components are introduced in the application graph, to model scheduling and resource allocation[1].

Throughput analysis with the conservative techniques involves multiple steps. 1) Constructing a $(max, +)$ matrix for each scenario; 2) Computing the cycle-time vector of the matrix of each scenario; 3) Computing the generalized eigenvector of the matrix of each scenario, as per Lemma 1; 4) Computing delay, as per Algorithm 3: a) the scenario graph and global initial vector approaches require delay computation for each scenario and b) the local initial vector approach requires delay computation for each FSM transition; 5) a MCM analysis on the

[1]This issue of resource-aware dataflow models is discussed further in Section 6.1.1

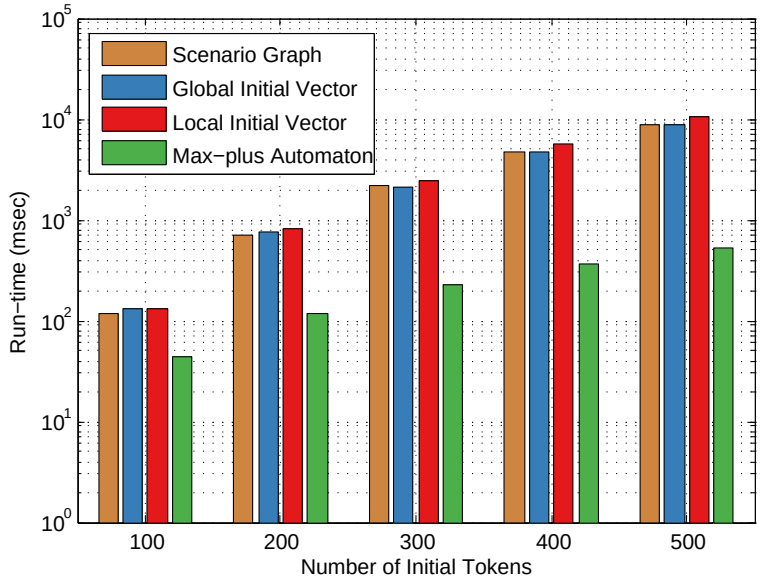


Figure 5.11: Analysis run-time vs. initial tokens (#FSM states=5)

FSM if the global initial vector and local initial vector approaches are used.

Constructing the $(max, +)$ matrix of a scenario (step 1) requires a symbolic execution of the scenario for one iteration, as per Algorithm 1 of [28]. Hence, it has complexity with the number of actor firings in one iteration. Steps 2 to 4 are mainly based on the recurrent execution $\gamma_{k+1} = M \cdot \gamma_k$, until a periodic phase is reached. Thus, they have a complexity of $\mathcal{O}(t \cdot n^2)$, where t is the length of the transient phase and $\mathcal{O}(n^2)$ is complexity of matrix-to-vector multiplication. However, the local initial vector technique requires transition delay computations (step 4) for each transition in the FSM. This makes the global initial vector technique computationally more expensive than the other two techniques. Step 5, which is a MCM analysis on the FSM, is not required for the scenario graph analysis technique (cf. Equation (5.17)). As a result, the technique is generally faster than the other two techniques.

The above discussion reveals that the conservative analysis techniques are dominated by matrix-to-vector multiplications. Hence, they have time-complexity which is polynomial with the number of initial tokens. The $(max, +)$ automaton technique is also affected by increasing number of initial tokens. However, its run-time may be low, even in case of a large number of initial tokens, if the scenario matrices are sparse (populated primarily with $-\infty$). This is illustrated in Figure 5.11, which shows analysis run-time for increasing number of initial to-

kens. The analysis is carried out using the FSM-SADF model of LTE, shown in Figure 3.4, by synthetically increasing the initial tokens on two channels: the channels from actor *dmem* to actor *odmp*, and from actor *chan* to actor $chef$. Since the resulting matrices are highly sparse, the number of edges of the throughput graph is much less than the maximum possible connections. Consequently, the exact technique is faster than the conservative techniques.

5.6.3 Conclusion

The evaluation in this section demonstrates that the presented analysis techniques can be directly applied to FSM-SADF graphs, which have non-self-timed bounded scenarios and inter-scenario synchronizations through non-common channels. Examples of such cases in Table 5.1 are the FSM-SADF graphs of Figure 5.2, WLAN and LTE. The results of the conservative techniques show that the local initial vector technique gives a better throughput bound than the other two techniques, while the scenario graph approach is the least accurate as it abstracts from scenario transitions. The improvements of the throughput bounds, however, come at additional run-time cost. The evaluation demonstrates the analysis techniques have time-complexity with the number of initial tokens and FSM states. The conservative techniques are in general faster than the $(max, +)$ automaton approach, although the latter may still perform better if the scenario matrices are sparse.

5.7 Related Work

Existing throughput analysis techniques of dataflow MoCs or similar analysis graphs in the literature predominately target Synchronous Dataflow (SDF) [53]. In particular, they focus on a sub-class of SDF where all port-rates are restricted to exactly one token, also known as Homogeneous SDF (HSDF). The throughput of a HSDF graph can be analysed through a maximum cycle mean (MCM) analysis [22] or through a $(max, +)$ formalization [8]. A detailed comparison of different MCM algorithms can be found in [21]. These MCM algorithms are in general applicable to SDF graphs through conversion to HSDF equivalents [27,81]. The conversion, however, can lead to an explosion in the size of the graph. As a result, it may negatively affect the performance of these algorithms, as demonstrated by the evaluation in [35]. To address this issue, [39] has proposed an alternative $(max, +)$-based throughput analysis approach using a linear constraint graph (LCG) that has fewer edges than an equivalent HSDF graph, by considering only SDF channels with the strongest data dependencies. Another solution presented in [35] is to directly analyse the throughput of self-timed bounded SDF

graphs, without requiring conversion to HSDF. The technique is based on a state-space exploration, where a state is defined by token distributions of channels and the remaining number of clock cycles of actor executions. The state-space execution of a self-timed bounded SDF graph reaches a periodic phase after a finite number of transient states. The periodic phase determines the throughput of the graph, in terms of the number of completed iterations per time-unit. Another SDF analysis technique that does not require conversion to HSDF is shown in [28]. The technique derives the worst-case throughput from the eigenvalue of the $(max, +)$ characterization matrix of the SDF graph (cf. Section 2.4). Nevertheless, as efforts being made to extend the expressiveness of SDF to support application dynamism, new dataflow MoCs have emerged (cf. Section 3.4), which called for appropriate support for these extensions during temporal analysis.

Variable-Rate Dataflow (VRDF) allows port-rates to vary within a specified range. Its extension Variable-Rate Phased Dataflow (VPDF) furthermore allows actors to cycle through a number of predetermined phases, each of which may execute multiple times, based on a run-time value selected from a finite interval [96]. However, [96] follows a different analysis approach for VRDF and VPDF than ours, as it does not provide a means to directly compute the worst-case throughput of these graphs (as we have shown for FSM-SADF in this chapter). Instead, it indirectly verifies whether a throughput constraint imposed by a source (a sink) actor can be satisfied by conservatively computing sufficient channel buffer sizes that enable the construction of strict periodic schedules for actor firings, along with their starting times.

Mode-Controlled Dataflow (MCDF) [59] supports modeling a dynamic streaming application by identifying different static modes, where each mode can be represented by a HSDF graph. [59] has proposed different techniques to perform worst-case throughput analysis of MCDF graphs. One simple technique is to derive a conservative throughput bound from the *rate-equivalent* HSDF graph, which is obtained by replacing all conditional productions and consumptions (i.e. switches and selects) of the MCDF graph by unconditional ones. This gives a bound that is guaranteed to be met by all modes, as the MCM analysis on the HSDF graph considers the MCDF graph as a whole. The other methods improve this conservative estimation either through a dataflow simulation of all mode sequences of interest or by linear-bounding mode sequences through strict periodic schedules of modes, along with appropriate mode transition intervals. However, the latter techniques have to be applied to all (finite number of) mode sequences of interest individually. As such, the throughput analysis abstracts from transitions between consecutive mode sequences. FSM-SADF makes use of a FSM to encode infinitely long and infinitely many scenario sequences. Our throughput analysis takes into account the pipelining between any two finite-length scenario-sequences,

which are executed, for example, to decode two consecutive data packets. This leads to tighter throughput bounds. Moreover, MCDF currently supports only HSDF modes, which have a scalability problem. Our analysis is performed directly on SDF scenarios, without requiring conversion to HSDF.

Scenario-Aware Dataflow (SADF) [89] improves the expressiveness of SDF by allowing variable port-rates and execution times of actors, while establishing correlations between different values of these parameters through the concept of scenarios. It uses a stochastic approach to model the orders in which scenarios occur. The throughput analysis of SADF presented in [89] constructs a global state-space representation of the execution of scenario sequences, where transitions are at the level of individual firings of actors. This may result in a very large state-space as the graph size grows.

The FSM-version of SADF, introduced in [28], encodes the possible orders of scenario executions using a non-deterministic FSM, thereby opening an opportunity to construct a much smaller state-space, whose transitions are at the level of scenario iterations [30]. The state-space enables to derive the exact worst-case throughput that may occur in the execution of the FSM-SADF. [30] also introduces another exact worst-case throughput analysis, known as $(max, +)$ automaton. Faster but conservative analysis of FSM-SADF have also been studied in [28, 69]. The conservative analysis approach is based on deriving reference schedules that bound scenario transitions, similar to our conservative worst-case throughput computation, discussed in Section 5.5.2.

However, these existing scenario-base analysis techniques require scenarios to be self-timed bounded. In case of self-timed unbounded scenarios, the spectral analysis presented in Algorithm 3 of [28] does not terminate, as a periodic phase would never be reached. Moreover, these works implicitly define inter-scenario synchronizations through common initial tokens. In this chapter, we analyse FSM-SADF without requiring self-timed boundedness of scenarios. Unlike SDF, this is essential since an unbounded scenario does not necessarily make a scenario sequence unbounded. In addition, we use identifiers of initial tokens, instead of implicit common initial tokens, to explicitly define data dependencies between scenarios.

Another extension to scenarios of FSM-SADF like ours is presented in [31] that permits individual scenarios to be inconsistent (cf. Definition 2) as long as scenario sequences within cycles of the FSM remain consistent. In this case, the boundedness analysis in Section 5.4 can be used to verify whether every scenario sequence within a cycle of the FSM is bounded using the characterization matrix of the scenario sequence. This matrix is constructed as a product of the matrices of the individual scenarios of the sequence (cf. Section 7.3.3 for more).

5.8 Summary

The individual scenarios of an FSM-SADF model should not necessarily be self-timed bounded. This is because a channel, which has a faster production rate than its consumption rate in one scenario, may stay bounded due to a slower production rate in another scenario. Moreover, scenarios should not necessarily have the same set of channels, as the graph structure of the application may change with the processed input stream. Existing FSM-SADF analysis techniques, however, are limited to 1) self-timed bounded scenarios and 2) inter-scenario synchronizations that are captured through initial tokens on common channels between scenarios. This chapter lifts these assumptions and presents a generalized approach to analyse the worst-case throughput of an arbitrary set of SDF scenarios.

The basis of the generalized analysis is an algorithm, presented in Section 5.4, that computes the rates of execution of the different self-timed bounded components of a scenario. The algorithm is based on the concept of generalized eigenmode from $(max, +)$ algebra [43]. The algorithm returns the cycle-time vector of the matrix of a scenario. The cycle-time vector verifies that a scenario is bounded if all entries of the vector are the same; since this implies that the rates of execution of all actors are matched. The algorithm can also be used to analyse boundedness of FSM-SADF models. In this case, boundedness needs to be verified for every cycle of the FSM, since any sequence of execution of scenarios is formed by concatenation of the cycles of the FSM. A cycle of the FSM is bounded if the matrix of the sequence of scenarios of the cycle is bounded. The matrix of a scenario sequence is basically the product of the matrices of the scenarios of the sequence.

Based on the results of the analysis of a single scenario in Section 5.4, existing worst-case throughput analysis techniques are generalized. These techniques have different levels of accuracy and run-time. The conservative techniques are generally faster and more scalable than the exact technique. However, they give only a lower-bound WCT whereas the exact technique gives the actual WCT that may occur in the execution of the FSM-SADF.

CHAPTER 6

Analysing Application Mappings

The previous chapter discussed the temporal analysis of FSM-SADF models, without explicitly considering the impact of resource sharing. This chapter presents an analytical method to compute temporal bounds of applications mapped onto a shared multi-core platform. The applications are modeled with the FSM-SADF MoC. The presented analysis method avoids constructing resource-aware dataflow models, which are often used in existing dataflow-based approaches. As a result, it keeps the graph size intact and improves scalability. The analysis method follows a new approach that combines symbolic simulation in $(max, +)$ algebra with *worst-case resource curves (WCRCs)*. A WCRC specifies the minimum amount of resource an application is guaranteed to get over a time interval. The analysis enables a tighter performance guarantee by improving the WCRTs of service requests that arrive in the same busy time of a resource.

The chapter is organized as follows. Section 6.1 motivates the mapping analysis problem and outlines our approach. Section 6.2 presents the problem formulation. Section 6.3 discusses the analysis of an FSM-SADF mapped on a multi-core platform. Section 6.4 discusses symbolic identification of busy times of a resource for improved WCRT analysis. Section 6.5 demonstrates derivation of WCRCs analytically for a real-time arbiter, namely CCSP. Section 6.6 evaluates the approach. Section 6.7 presents related work. Section 6.8 summarizes the chapter.

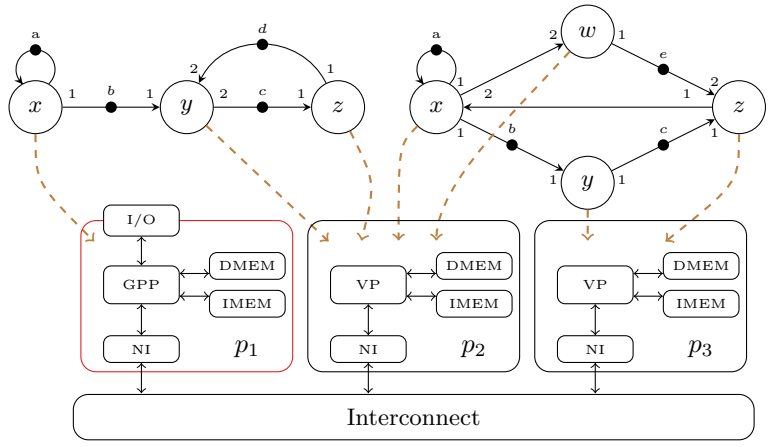


Figure 6.1: Example SDF scenarios mapped onto a MPSoC

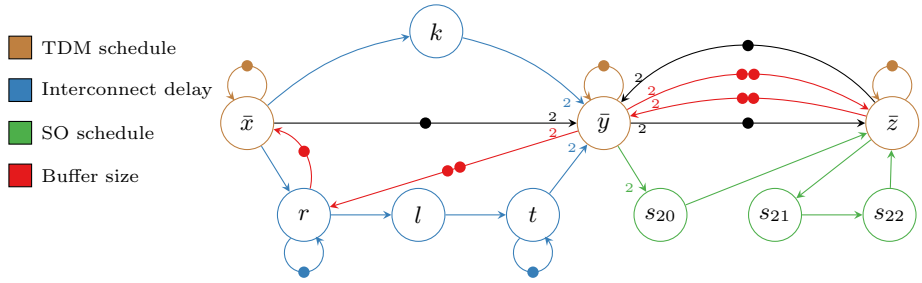


Figure 6.2: A RAD model of graph g_2 of Figure 6.1

6.1 Introduction

Embedded multimedia and wireless systems are typically implemented on a multiprocessor system-on-chip (MPSoC) that comprises heterogeneous resources that are shared between multiple applications under different scheduling policies. These applications have strict real-time constraints, such as throughput and latency. It is crucial to guarantee that such constraints are satisfied at all times. Analytical approaches have been widely used to compute conservative temporal bounds (cf. related works in Section 6.7). Analytical approaches should give tight bounds to avoid unnecessary resource over-allocation. They should also be fast to explore the mapping design-space efficiently through an iterative process, following the Y-chart approach (cf. Figure 1.2). Analysing application mappings using dataflow MoCs [59, 86] is among such analytical approaches that can give conservative temporal guarantees at design-time.

Dataflow-based mapping analysis approaches [59, 81, 86] construct a *resource-aware dataflow (RAD)* model to analyse application mappings. A RAD model is generated by adding mapping-related dataflow components into the application model [20] [55]. These components may dilate the graph and result in poor scalability, as discussed in Section 6.1.1. Besides, RAD supports only schedulers that have dataflow representations. This chapter presents a new analysis approach, which addresses these limitations, as outlined in Section 6.1.2.

6.1.1 Motivation

Existing dataflow-based analysis approaches [86] [98] [59] construct a RAD model after an application is mapped on a MPSoC platform. The RAD model incorporates different system aspects. Figure 6.1 shows two SDF graphs that are mapped on a heterogeneous MPSoC platform model that comprises a general purpose processor (GPP p_1) and two vector processors (VP p_2 and p_3). Figure 6.2 shows the RAD model of graph g_2 of Figure 6.1. A static-order (SO) schedule is used between actors of the graph. A budget scheduler (e.g time-division multiplexing (TDM)) is used to share the platform with other graphs.

In the RAD model, the buffer-size of a channel is modeled by adding another channel in the reverse direction (red edges), with as many initial tokens as the allocated buffer-size [68]. The model of the SO schedule (green actors and edges) is based on [20]. The model of the communication delay from actor x to y (blue) is based on the dataflow model for a class of schedulers known as $\mathcal{LR}$ servers [98]. The TDM scheduling is modeled by replacing each actor x by another actor $\bar{x}$ (brown), whose execution time equals the worst-case response time (WCRT) of x. The WCRT captures the maximum interference from actors of other graphs. This WCRT model can be refined by replacing each actor by the two-actor latency-rate ($\mathcal{LR}$) model [98] or even by the *many-actor* model of [55].

The total number N of initial tokens of a RAD model plays a key role in the run-time of temporal analysis. Each matrix has a dimension of $N \times N$ and the throughput analysis has time complexity with the number of initial tokens, as discussed in Chapter 5. Figure 6.3 shows the number of initial tokens of RAD models for different applications, after mapping onto a 4-tile multi-core target using the SDF3 tool [86]. The applications are taken from the benchmark graphs of [86]. The figure shows that the throughput analysis takes only few seconds for $N < 500$. When N grows to few thousands, the run-time increased to more than 10 minutes. For the last 3 cases, the large numbers of tokens resulted in huge matrices, which led to running out of memory and the analysis could not be completed. As a result, techniques to limit the number of initial tokens to a manageable level become crucial.

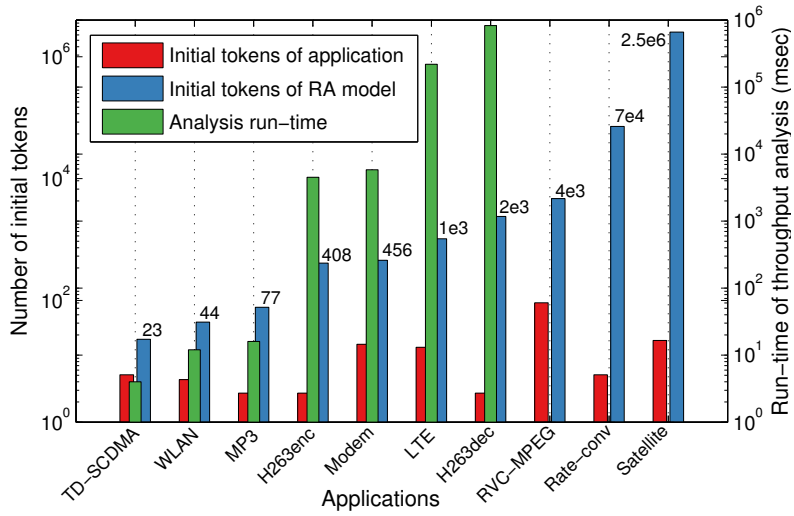


Figure 6.3: Scalability challenge in RAD-based analysis.

6.1.2 Outline of the Approach

This chapter presents an approach, which we call *symbolic analysis of application mappings (SAAM)*, to analyse application mappings, without explicitly constructing RAD models. The approach uses FSM-SADF to model applications. In FSM-SADF, each scenario is modeled by a SDF graph. The resulting set of scenarios are analysed compositionally after analysing each scenario separately, using the techniques discussed in Chapter 5. A scenario is analysed by constructing a characterization matrix of the corresponding SDF graph. Existing dataflow-based analysis techniques construct such matrices after RAD models are constructed. This approach may become problematic as the graph size grows (cf. Section 6.1.1). This chapter proposes an approach that embeds worst-case resource curves in the dataflow analysis to model the worst-case (minimal) availability of resources. The dataflow analysis is based on a symbolic simulation in $(max, +)$ algebra [8], which takes the static-order schedule of actors and shared resource contentions into account. As a result, the analysis keeps the graph size, in particular the number of initial tokens, intact and enables a better scalable analysis. The technique is illustrated for four different types of resource models: buffers, static-order scheduling, budget scheduling and interconnects. The technique further enables tighter temporal bounds by improving the WCRTs of requests that arrive in the same *busy time* (i.e. an interval between two idle times) of a resource.

6.1.3 Contribution

The chapter presents three novel contributions: 1) a matrix characterization of a dataflow mapping without explicitly constructing a RAD model, and 2) embedding worst-case resource curves during a $(max, +)$ symbolic simulation to characterize resource scheduling and 3) symbolic identification of busy times for improved WCRT analysis. The first two contributions, which are presented in Section 6.3, keep the graph size intact and enable a scalable analysis. The third contribution, which is presented in Section 6.4, enables tighter temporal bounds by avoiding the pessimistic assumption of *critical instant* on all requests. The critical instant of a request in TDM is, for instance, when a request arrives just after its allocated time slot has passed.

The new analysis approach is evaluated in Section 6.6 with different multimedia and wireless application graphs. The results show significant improvements in analysis run-time and tightness of temporal guarantees.

6.2 Problem Formulation

This chapter intends to compute worst-case temporal bounds for a streaming application that is mapped on a heterogeneous set of resources that are shared with other applications. The application model is the FSM-based SADF MoC, which is defined in Chapter 2. The resource model and the application-to-resource mapping are discussed below.

6.2.1 Resource Model

The MPSoC platform comprises a set Π of heterogeneous processor tiles. The interconnect is abstracted with a set Θ of connections. The platform (Π, Θ) can be shared between multiple applications under different scheduling policies. Mapping an application (S, f) to platform (Π, Θ) allocates resources such as processor budgets. For instance, on a processor under TDM scheduling, an application is allocated a slice (say 25%) of the TDM frame. At run-time, the application is preempted by other applications when it runs out of its allocated slice and has to wait for the next TDM frame.

In general, we characterize the minimum share an application (S, f) has on a platform (Π, Θ) with *worst-case resource curves (WCRCs)*. WCRCs are the same as lower-bound service curves of real-time calculus [17]. A WCRC $\xi(t_\delta)$, defined in Definition 15, specifies the minimum amount of resource in *service units* that an application is guaranteed to get in a given *time interval* $t_\delta \in \mathbb{N}$ (where $t_\delta = v - u$ for any time interval $[u, v)$, $u, v \in \mathbb{N}$).

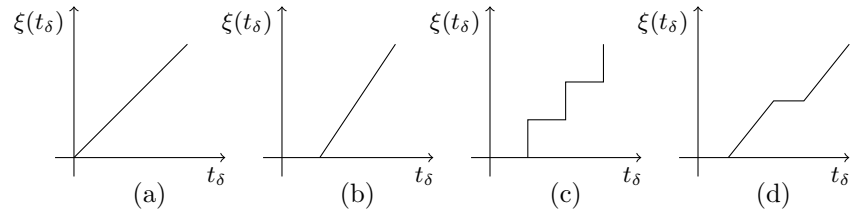


Figure 6.4: Example WCRCs

Service unit can be, for example, processor time in cycles for an actor execution or connection bandwidth in bytes for a token transaction. Figure 6.4 shows some examples of WCRCs, which show (a) a fully available resource, (b) a bounded delay resource that guarantees a certain service rate after a bounded delay, (c) a periodic resource that periodically processes a request and (d) a TDM resource that replenishes the allocated slice every frame.

Definition 15 (Worst-case Resource Curve - WCRC)**.** *A WCRC ξ conservatively models an actual resource r if for any time interval $[u, v) : u, v \in \mathbb{N}, u \leq v$, if $r[u, v)$ denotes the amount of service units that an application is guaranteed to get from the resource over interval $[u, v)$, then $r[u, v) \geq \xi(v - u)$.*

WCRCs capture resource sharing between multiple applications. This is because each curve specifies the minimal service an application is guaranteed to get from a processor or a connection, irrespective of other applications sharing the resource. As a demonstration, Section 6.5 analytically derives the WCRC for one real-time scheduler, namely Credit-Controller Static Priority (CCSP) [6], often used in predictable memory controllers.

6.2.2 Application Mapping

The application-to-platform mapping decides actor-to-processor and channel-to-connection or channel-to-tile bindings. In addition, it allocates resources such as processor budgets and buffer-sizes. It also constructs a SO schedule between actors (of the same graph) that are mapped on the same processor. The mapping of an application may vary from one scenario to another. Definitions 16 defines a scenario mapping, where Ξ denotes the set of WCRC functions.

Definition 16 (Scenario mapping)**.** *Given a platform* (Π, Θ)*, a scenario mapping* $(g, \tau, \kappa, \pi, \theta, \beta, \sigma)$ *is a 7-tuple, where* g *is a scenario graph,* $\tau : A \rightarrow \Pi$ *is actor-to-processor binding and* $\kappa : C \rightarrow \Theta \cup \Pi$ *is channel binding.* $\pi : \Pi \rightarrow \Xi$ *and* $\theta : \Theta \rightarrow \Xi$ *are the worst-case resource curves of processors and connections, respectively.* $\beta : C \rightarrow \mathbb{N}$ *is the allocated buffer-sizes of channels.* σ *is a function that returns the SO schedule* $\sigma(p) = \langle a_1, a_2, \cdots, a_n \rangle$ *of actors* $a_i \in A$ *on processor tile* $p \in \Pi$*.*

An application mapping (S, f, μ) is a 3-tuple. S is a set of scenarios, f is a FSM on S and μ is a function that gives the scenario mapping $\mu(s)$ of $s \in S$. We present our mapping analysis technique in two steps. First, Section 6.3 analyzes an application mapping by constructing a $(max, +)$ characterization matrix for each scenario mapping. Then, Section 6.4 improves the tightness/accuracy of the analysis by computing tighter WCRT bounds for requests that arrive in the same busy time of a resource.

6.3 Matrix Characterization of a Mapping

Given a scenario mapping, an actor is enabled for firing if 1) the actor is next in the SO schedule, 2) all input tokens have arrived and 3) there is sufficient output buffer space. Thus, the enabling time of an actor is determined by the last condition to be satisfied; i.e. the *maximum* of the enabling times of the above three conditions. An upper-bound to the completion time of an actor's firing can be obtained by *adding* its WCRT to its start time, as shown in Equation 6.1. $\phi(x, k)$ denotes the upper-bound for the completion time of the k^{th} firing of actor x. The parameters t_i, t_o, t_p denote input token, output buffer and processor availability times, respectively.

$$\phi(x, k) = \max(t_i, t_o, t_p) + \omega(x) \tag{6.1}$$

$\omega(x)$ denotes the WCRT of actor x. It depends on the scheduling policy used to share the processor tile with other applications. It is computed from the WCRC ξ of the tile, as shown in Equation (6.2). Recall that $\chi(x)$ is the WCET of the actor (cf. Definition 1). Equation (6.2) returns the minimum time interval that is needed to obtain sufficient service cycles for the WCET of the actor.

$$\omega(x) = \inf\{t_\delta \in \mathbb{N} \mid \xi(t_\delta) \geq \chi(x)\}. \tag{6.2}$$

Equation (6.1) illustrates how the timing behavior of a scenario mapping can be analyzed using $(max, +)$ expressions. The parameters t_i, t_o and t_p can also be derived through $(max, +)$ expressions as explained next.

(a) c_{xy} has n initial tokens (b) c_{xy} has a buffer-size of $\beta(c_{yx}) = m$

Figure 6.5: Modeling the allocated buffer-size of a channel in RAD

1. The input data availability time t_i is determined by the last arriving token, i.e. the maximum of the production times of all input tokens.

2. The processor availability time t_p can be derived from the completion time of the last actor that is executed on the processor.

3. Output buffer availability time t_o can be computed from the completion times of previous actor firings. This is because we employ an *acquire-release* FIFO management, where buffer spaces of input tokens are conservatively released only after the actor firing is completed. E.g. Figure 6.5(a) shows channel c_{xy} from actor x to y with $n \in \mathbb{N}^0$ initial tokens. The channel is allocated a buffer-size of $m \in \mathbb{N}$ where $m \geq n$. The availability times of free buffer spaces on this channel can be derived by considering how buffer spaces are modeled in RAD. Buffer spaces are modeled in RAD by adding another channel in the reverse direction with as many initial tokens as the number of free buffer spaces [68], as shown in Figure 6.5(b). The tokens on the reverse channel c_{yx} represent the released/free buffer spaces. The output buffer availability time on channel c_{xy} for the k^{th} firing of actor x is given by Equation (6.3). The availability time is basically obtained from the completion time of a previous firing of actor y.

$$\psi(x, k, c_{xy}) = \phi\left(y, \left\lceil \frac{p \times k - m + n}{q} \right\rceil\right) \tag{6.3}$$

The output buffer availability time $t_o = \psi(x, k)$ of x for the k^{th} firing is then the latest buffer space availability time among all outgoing channels of x, i.e. $C_{xy} = \{c_{xy} \in C \mid y \in A, c_{xy} = (x, y)\}$, as shown in Equation 6.4.

$$\psi(x, k) = \max_{c_{xy} \in C_{xy}} \psi(x, k, c_{xy}) \tag{6.4}$$

Example: We show the execution of the scenario mapping of graph g_2 of Figure 6.1. Assume the WCETs of the actors are $\chi(x) = 2, \chi(y) = 3$ and $\chi(z) = 1$. Actor x is mapped on processor tile p_1 and actors y and z are mapped on p_2. The allocated buffer sizes of channels are $\beta(c_{xy}) = 1$, $\beta(c_{yz}) = 3$ and $\beta(c_{zy}) = 2$. The buffers allow the SO schedules $\sigma(p_1) = \langle x \rangle$ and $\sigma(p_2) = \langle z, y, z \rangle$. We assume TDM scheduling on both tiles p_1 and p_2 with a TDM frame size of 4 and allocated TDM slice of 2 time-units. To simplify the example, we assume delay-less interconnect. Interconnect delays are discussed later in Section 6.3.2.

An execution of one iteration comprises four actor firings as given by the repetition vector $[1, 1, 2]$. Initially, the only possible firing is actor z. Actor x cannot fire since it does not have enough output buffer space and actor y is not next in the SO schedule. When actor z fires, it consumes token c and produces one token on channel c_{zy}. The firing completes after its WCRT $\omega(z) = 3$, waiting for a maximum of 2 time-units followed by execution of 1. (The computation of the WCRTs on tile p_2 can also be seen on Figure 6.8, which shows the WCRC of a TDM of frame size 4 and allocated slice 2.) It then produces one token on channel c_{zy}, time-stamped with its production time 3. Then, actor y fires and completes its firing at $\phi(y, 1) = \max(t_i, \psi(y, 1), t_p) + \omega(y) = \max(3, 0, 3) + 7 = 10$. Note that $\psi(y, 1) = 0$ since channel c_{yz} had already 2 free buffer spaces at the beginning of the iteration. This can also be derived from Equation (6.4), which gives $\psi(y, 1) = \phi(z, 0) = 0$. Similarly, the remaining two firings complete at $\phi(x, 1) = 10 + 4 = 14$ and $\phi(z, 2) = 10 + 3 = 13$. The collection of the time-stamps of the final tokens marks a bound to the end of the iteration. This enables temporal analysis of a scenario mapping such as throughput, which is given by iterations per time-unit, as discussed in Chapter 5.

6.3.1 $(max, +)$ matrix of a scenario mapping

Next, we execute the above example using *symbolic time-stamps* of tokens. By symbolic execution, as opposed to concrete execution like the above example, we provide a means to compute a bound for any iteration, given the collection of time-stamps that mark the start of the iteration. This is given by a recurrent relation $\bar{\gamma}_{k+1} = M \cdot \bar{\gamma}_k$, where time-stamp vector $\bar{\gamma}_k \in \mathbb{R}^n_{\max}$ marks the end of iteration k. $M \in \mathbb{R}^{n \times n}_{\max}$ is a $(max, +)$ characterization matrix of the scenario mapping.

The production time of a token can be put symbolically as $t = \max_i(t_i + g_i)$, where t_i denotes time-stamps of initial tokens and resource availability, and g_i denotes suitable constants, as presented in Section 2.4. This can be written in a $(max, +)$ vector dot-product $\bar{t}.\bar{g}$, where $\bar{t} = [t_a, t_b, t_c, t_d, t_{p1}, t_{p2}]^T$ and $\bar{g} \in \mathbb{R}^6_{\max}$. The time-stamp vector has 6 entries: 4 for the initial tokens (a, b, c and d) and 2 for processors p_1 and p_2.

Let the time-stamps $\bar{t}_a, \bar{t}_b, \bar{t}_c$ and $\bar{t}_d$ correspond to the following time-stamp vectors, respectively: $[0; -\infty; -\infty; -\infty; -\infty; -\infty]^T$, $[-\infty; 0; -\infty; -\infty; -\infty; -\infty]^T$, $[-\infty; -\infty; 0; -\infty; -\infty; -\infty]^T$ and $[-\infty; -\infty; -\infty; 0; -\infty; -\infty]^T$. Similarly, the time-stamps $\bar{t}_{p1}$ and $\bar{t}_{p2}$ encode the availability times of the two processors (p_1 and p_2). They correspond to the time-stamp vectors $[-\infty; -\infty; -\infty; -\infty; 0; -\infty]^T$ and $[-\infty; -\infty; -\infty; -\infty; -\infty; 0]^T$. Using symbolic versions of Equation (6.1) and (6.4), where all time-stamps are vectors, the completion time of the first firing of actor z is given by Equation (6.5).

$$\begin{aligned} \bar{\phi}(z,1) &= \max(\bar{t}_c, \bar{\psi}(z,1), \bar{t}_{p2}) + \omega(z) \\ &= \max(\bar{t}_c, \bar{t}_{p2}) + \omega(z) \\ &= [-\infty; -\infty; 0; -\infty; -\infty; 0] + 3 \\ &= [-\infty; -\infty; 3; -\infty; -\infty; 3] \end{aligned} \tag{6.5}$$

Note that $\bar{t}_c$, $\bar{\psi}(z,1)$ and $\bar{t}_{p2}$ are the input token, output buffer and processor availability times, respectively. According to Equation (6.4), $\bar{\psi}(z,1)$ evaluates to $\bar{\phi}(y,0)$, which is the completion time of actor y in the previous iteration on tile p_2. Hence, $\bar{\psi}(z,1)$ occurs before the current availability time of p_2: i.e. $\bar{\psi}(z,1) \preceq \bar{t}_{p2}$. This simplifies the evaluation as shown in Equation (6.5). After the firing of actor z, the symbolic time-stamp of processor p_2, $\bar{t}_{p2}$ is updated to $\bar{\phi}(z,1)$, marking the availability of the processor for the next actor firing. This way, the static-order scheduling of actors mapped on the same processor tile is properly taken into account in the mapping analysis.

Similarly, we have the following completion times for the other firings.

- The firing of actor y

$$\begin{aligned} \bar{\phi}(y,1) &= \max(\bar{t}_b, \max(\bar{t}_d, \bar{\phi}(z,1)), \bar{t}_{p2}, \bar{\psi}(y,1)) + \omega(y) \\ &= \max(\bar{t}_b, \max(\bar{t}_d, \bar{\phi}(z,1)), \bar{\phi}(z,1), \bar{\psi}(y,1)) + \omega(y) \\ &= [-\infty; 0; 3; 0; -\infty; 3] + 7 \\ &= [-\infty; 7; 10; 7; -\infty; 10] \end{aligned}$$

  After the completion of actor y, $\bar{t}_{p2}$ is updated to $\bar{\phi}(y,1)$.

- The second firing of actor z

$$\begin{aligned} \bar{\phi}(z,2) &= \max(\bar{\phi}(y,1), \bar{t}_{p2}, \bar{\psi}(z,2)) + \omega(z) \\ &= \max(\bar{\phi}(y,1), \bar{\phi}(y,1), \bar{\psi}(z,2)) + 3 \\ &= [-\infty; 10; 13; 10; -\infty; 13] \end{aligned}$$

  After the completion of the second firing of actor z, $\bar{t}_{p2}$ is updated to $\bar{\phi}(z,2)$.

- The firing of actor x

$$\begin{aligned}\bar{\phi}(x,1) &= \max(\bar{t}_a, \bar{t}_{p1}, \bar{\psi}(x,1)) + \omega(x) \\ &= \max(\bar{t}_a, \bar{t}_{p1}, \bar{\phi}(y,1)) + 4 \\ &= [0;7;10;7;0;10] + 4 \\ &= [4;11;14;11;4;14]\end{aligned}$$

After the completion of actor x, $\bar{t}_{p1}$ is updated to $\bar{\phi}(x,1)$.

Thus, the completion times of the firings are $\bar{\phi}(y,1) = [-\infty;7;10;7;-\infty;10]$, $\bar{\phi}(z,2) = [-\infty;10;13;10;-\infty;13]$ and $\bar{\phi}(x,1) = [4;11;14;11;4;14]$. At the end of the iteration, the new four tokens have the time-stamps $\bar{t'}_a = \bar{\phi}(x,1)$, $\bar{t'}_b = \bar{\phi}(x,1)$, $\bar{t'}_c = \bar{\phi}(y,1)$ and $\bar{t'}_d = \bar{\phi}(z,2)$. The time-stamps of the two processors become $\bar{t'}_{p1} = \bar{\phi}(x,1)$ and $\bar{t'}_{p2} = \bar{\phi}(z,2)$. Collecting the new symbolic time-stamps as $[\bar{t'}_a, \bar{t'}_b, \bar{t'}_c, \bar{t'}_d, \bar{t'}_{p1}, \bar{t'}_{p2}]$ gives Equation (6.6), which captures the worst-case timing behavior of an iteration. The relation enables to compute a bound for any iteration, given the start of the iteration, as per the relation $\bar{\gamma}_{k+1} = M \cdot \bar{\gamma}_k$.

$$\begin{bmatrix} t'_a \\ t'_b \\ t'_c \\ t'_d \\ t'_{p1} \\ t'_{p2} \end{bmatrix} = \begin{bmatrix} 4 & 11 & 14 & 11 & 4 & 14 \\ 4 & 11 & 14 & 11 & 4 & 14 \\ -\infty & 7 & 10 & 7 & -\infty & 10 \\ -\infty & 10 & 13 & 10 & -\infty & 13 \\ 4 & 11 & 14 & 11 & 4 & 14 \\ -\infty & 10 & 13 & 10 & -\infty & 13 \end{bmatrix} \cdot \begin{bmatrix} t_a \\ t_b \\ t_c \\ t_d \\ t_{p1} \\ t_{p2} \end{bmatrix} \tag{6.6}$$

6.3.2 Accounting for Interconnect Delay

If both the source and destination actors of a channel c are mapped on the same tile, the FIFO buffer of the channel is allocated locally on the tile; i.e. $\kappa(c) \in \Pi$. In this case, modeling these local buffers with time-stamp tokens is not required, since there is no overlapping between different scenarios of an application on the same tile. The completion time of a scenario on a tile can be found solely from the tile's time-stamp (i.e. $\bar{t}_{p_1}$ and $\bar{t}_{p_2}$ of the running example). This is basically why we had the simplification earlier in Equation (6.5).

If, however, a channel is mapped on a connection, the earlier simplification no longer holds. Due to the possibility of overlapped execution of scenarios of an application on different tiles, we need additional time-stamp tokens for inter-tile connections in our matrix characterization. These tokens are required to carry the availability times of connections, source buffers and destination buffers between consecutive iterations. This section shows how interconnect delays are properly accounted for in the mapping analysis.

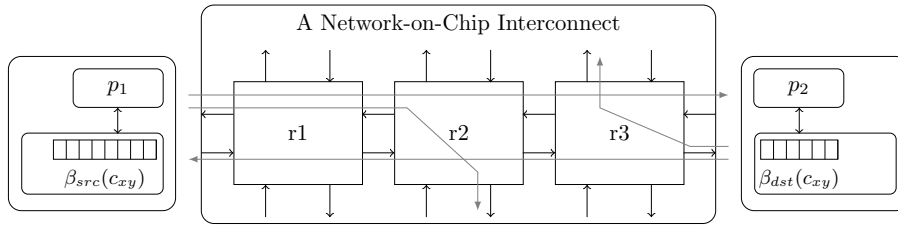
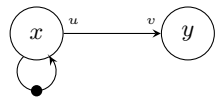
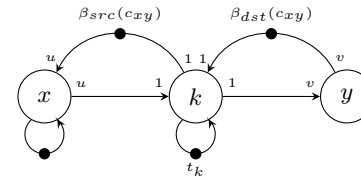


Figure 6.6: The buffer-size $\beta(c_{xy})$ of channel c_{xy} mapped on a connection is distributed on the source and destination tiles as $\beta(c_{xy}) = \beta_{src}(c_{xy}) + \beta_{dst}(c_{xy})$



(a) assume c_{xy} is mapped on a connection; i.e. $\kappa(c_{xy}) \in \Theta$



(b) connection $\kappa(c_{xy})$ is modeled by a single actor k with a self-edge

Figure 6.7: Accounting for inter-tile connection delays

If the source and destination actors x and y of channel c_{xy}, with port-rates $\rho(c_{xy}) = (u, v)$, are mapped on different tiles (Figure 6.7(a)), then the channel is bound to a connection $\kappa(c_{xy}) \in \Theta$. In this case, the allocated buffer-size $\beta(c_{xy})$ of the channel is distributed on the source and destination tiles, say $\beta_{src}(c_{xy})$ and $\beta_{dst}(c_{xy})$, as shown in Figure 6.6. A token that is to be sent over the connection is first written in $\beta_{src}(c_{xy})$. If a buffer space is available at the destination, the token is sent, which takes a certain transfer delay before it is written in $\beta_{dst}(c_{xy})$.

One approach to account for connections in our mapping analysis is to use existing RAD techniques to model transfer delays and buffers into the application graph, and thereafter, use the resulting partial-RAD model as a starting point for the analysis. The inter-tile connection can be modeled in RAD by a single dataflow actor or by a two-actors latency-rate model (as shown in Figure 6.2) or even by a detailed interconnect model [42]. In such cases, the matrix characterization of Section 6.3.1, as it is, properly takes into account interconnect delays.

An alternative approach is to use our WCRC characterization of resources for connections. For this approach, a single actor k is introduced between actors x and y, as shown in Figure 6.7(b). The WCRT of actor k is computed from the WCRC of the connection, i.e. $\theta(\kappa(c_{xy}))$, as given by Equation (6.2). It gives the transfer

delay. Actor k has a self-edge with one initial token t_k that stores the availability time of the connection after the previous transaction is completed. This way, one time-stamp token is introduced per connection to capture its availability time, similar to what we did for processor tiles.

The modeling of source and destination buffers is done the same way whether the RAD-based or the WCRC-based connection analysis is used. The buffers are modeled by adding reverse channels with as many initial tokens as the allocated buffer sizes [68], as shown in Figure 6.7. This means the source and destination buffers require one time-stamp token for each buffer space. However, the number of time-stamp tokens can be reduced, by taking into account the multi-rate data consumption and production behavior of SDF actors. In Figure 6.7(b), all $v \in \mathbb{N}$ buffer spaces released by the firing of actor y (i.e. tokens produced on channel c_{yk}) have the same time-stamp. The output buffer availability time of actor x is determined by the last claimed space (i.e. the last token among the $u \in \mathbb{N}$ tokens consumed from channel c_{kx}). As a result, for the source buffer $\beta_{src}(c)$, keeping only $\lceil \beta_{src}(c)/u \rceil$ time-stamp tokens in the matrix is sufficient, since the availability time of u consecutive buffer spaces needed by a firing of actor x is obtained from the last space released among these u spaces. Similarly, for the destination buffer $\beta_{dst}(c)$, introducing $\lceil \beta_{dst}(c)/v \rceil$ time-stamp tokens is sufficient, since the release times of v buffer spaces by a firing of actor y are all the same.

We conclude the section with some remarks on the RAD-based and WCRC-based connection analyses. As far as scalability is concerned, the WCRC-based analysis does not bring improvements if the intention is to substitute simple RAD models, such as single actor and latency-rate models. Its strong suit is its genericity. It does not require a dataflow model to be available for connections. It enables to compute WCRTs (in this case transfer delays), given the minimum amount of bytes a connection is guaranteed to transfer over different time-intervals. The other point is accuracy. Obviously, it gives the same WCRT as a single-actor RAD connection model. However, it gives pessimistic WCRT bounds compared to a latency-rate model, since the equivalent bounded-delay WCRC (cf. Figure 6.4b) of actor k assumes the initial delay/latency for every transfer request when the WCRT is computed using Equation (6.2). In the latency-rate model, the latency is seen only by the first request in case of a bursty transfer request (i.e. for a burst of n requests, the response-time is $(latency + \frac{1}{rate}) + \frac{n-1}{rate}$). In Section 6.4, we address this pessimism of WCRC-based WCRT computation of Equation (6.2) applying the concept of busy times [73]. The technique identifies the busy times of a resource from the symbolic time-stamps of actor firings. It then computes a WCRT based on the cumulative requested service since the beginning of a busy time. This enables to avoid assuming the initial delay for every transfer request and compute equivalent WCRTs as the latency-rate model.

Algorithm 4 Construct a $(max, +)$ matrix of a scenario mapping

1: ConstructMaxPlusMatrix $(g, \tau, \kappa, \pi, \theta, \beta, \sigma)$
2: $\Gamma \leftarrow$ the set of initial tokens of $g = (A, C, \iota, \chi, \rho)$
3: $T \leftarrow \{\bar{t}_k \mid \bar{t}_k$ is a symbolic token for each $k \in \Pi, \Omega\}$
4: $\bar{r}' \leftarrow$ compute the repetition vector of g
5: $\bar{r} \leftarrow \bar{r}'$
6: **while** $\bar{r} \neq$ a zero vector **do**
7: $\quad x \leftarrow$ pick an enabled actor $\mid \bar{r}(x) > 0$
8: $\quad C_i \leftarrow \{c_i \in C \mid c_i = (a, x) : a \in A, \rho(c_i) = (u_i, v_i)\}$
9: $\quad C_o \leftarrow \{c_o \in C \mid c_o = (x, a) : a \in A, \rho(c_o) = (u_o, v_o)\}$
10: $\quad T_i \leftarrow$ collect $v_i \in \mathbb{N}$ tokens from $\forall c_i \in C_i$
11: $\quad \bar{t}_{\tau(x)} \leftarrow$ pick the symbolic token of $\tau(x)$ from T
12: $\quad \bar{t}_o \leftarrow \max\left(\max_i T_i, \psi(x, \bar{r}'(x) - \bar{r}(x)), \bar{t}_{\tau(x)}\right) + \omega(x)$
13: $\quad T_o \leftarrow$ produce $u_o \in \mathbb{N}$ tokens at time $\bar{t}_o$ for $\forall c_o \in C_o$
14: $\quad T \leftarrow (T \backslash T_i) \cup T_o$
15: $\quad \bar{t}_{\tau(x)} \leftarrow \bar{t}_o$
16: $\quad \bar{r}(x) \leftarrow \bar{r}(x) - 1$ *//decrement number of firings*
17: $\quad$ update $\sigma(\tau(x))$ *//update SO schedule*
18: **end while**
19: $M \leftarrow$ collect all symbolic tokens $\bar{t}_k \in T$

6.3.3 Matrix Construction Algorithm

Algorithm 4 sketches the construction of a $(max, +)$ matrix of a given scenario mapping. To simplify the presentation of the algorithm, we assume inter-tile connections are already modeled in the scenario graph, as illustrated in the dataflow model of Figure 6.6. Hence, tokens that capture the availability times of connections as well as source and destination buffers are already in the set of initial tokens of the scenario graph.

The algorithm begins with associating a symbolic token $\bar{t}_k$ for each initial token and each processor tile. These symbolic tokens are collected into the set T (line 3). Then, it computes the number of actor firings in one iteration of the scenario (line 4-5). The algorithm picks an enabled actor (line 7). C_i and C_o are the set of input and output channels of the actor (line 8-9). The algorithm then computes a bound $\bar{t}_o$ to the completion time of the actor firing (line 12), by adding the WCRT of the actor to its enabling time. The enabling time is the maximum of the availability times of 1) all input tokens T_i (line 10), 2) output buffers and 3) the processor tile (line 11). The output buffer availability $\psi(x, k)$ of the k^{th} firing of x, where $k = \bar{r}'(x) - \bar{r}(x)$, is computed from previous actor

firings (cf. Equation (6.4)). At the end of the execution, new symbolic tokens are produced, all time-stamped at $\bar{t}_o$ (line 13). The symbolic execution terminates when all actors are executed as many times as their repetition factors. At the end of the execution, the set T contains the same number of tokens as the initial set; exactly one token for every initial token of the scenario graph and all tokens that model resource availability updated with new values. The collection of these tokens forms the $(max, +)$ matrix M (line 19) that characterizes the execution of the scenario mapping.

6.3.4 Composing Matrices of Scenario Mappings

Once a matrix $M_{\mu(s)}$ is constructed for each scenario mapping $\mu(s)$ of $s \in S$, the compositional analysis approach of Chapter 5 is used to analyse the application mapping (S, f, μ). Both conservative and exact analyses are possible. A simple conservative analysis is done by assuming a fully-connected FSM [30]. This implies that all scenario sequences are possible. In this case, the WCT equals the inverse of the eigenvalue of the matrix of Equation 6.7.

$$M = \max_{s \in S} M_{\mu(s)} \tag{6.7}$$

Another conservative approach is a maximum-cycle-mean analysis on the FSM f, since all scenario sequences are basically formed by concatenation of the cycles of the FSM. Such conservative techniques are discussed in Section 5.5.2. For the exact WCT, the $(max, +)$-automaton approach of [30] can be used.

6.4 Improving Response-time

We refer to a *service request* as an abstract unit of granularity of resource access, such as an actor firing on a processor, a data transaction on an interconnect or a read/write request to a memory controller. In Section 6.3, WCRCs are used to derive worst-case response bounds of individual service requests, as given by Equation (6.2). This is, however, pessimistic, since it assumes the *critical instant* irrespective of the arrival time of a request to a resource [73]. The critical instant can be the worst-case interference in priority-based scheduling or the arrival of a request just after the end of its allocated slot in TDM scheduling. A known approach to address this pessimism is the concept of *multiple event busy time* [73] [92]. A busy time refers to an interval between two idle times of a resource. The basic idea of this approach is to use the arrival time of a request to check if it has arrived while the resource is busy processing previously arrived requests so that the assumption of the critical instant can be avoided for the request. The

idea has been used in the literature for tighter response time analysis [92] and improved event model construction [73]. In this section, we apply the concept to improve WCRTs. Our contribution is a symbolic identification of busy times so that bounds tighter than Equation (6.1) can be derived. By using symbolic analysis, we make it possible to apply the technique by identifying all busy times of an iteration from the time-stamps of requests.

A busy time is a maximum time interval during which all requests except the first one have arrived before the preceding is finished [73]. Figure 6.8 illustrates an example, where the WCRC of a TDM arbiter of frame size 4 time-units and 50% slice allocation is shown. $\alpha(k)$ denotes the arrival time of request k and $\sigma(k)$ denotes the total amount of service requested in the busy time until and including k. The figure shows three request arrivals at time $\alpha(1) = 0$, $\alpha(2) = 2$ and $\alpha(3) = 7$, each of which individually request a service of 1, 2 and 1 service units, respectively. Hence, the total requested service until request 1 is $\sigma(1) = 1$, until request 2 is $\sigma(2) = 1 + 2 = 3$ and until request 3 is $\sigma(3) = 1 + 2 + 1 = 4$.

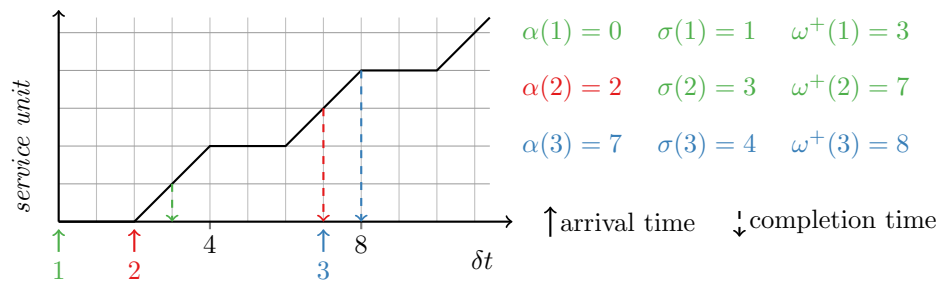


Figure 6.8: Example WCRT analysis using busy times.

In our analysis, we compute $\omega^+(\sigma) \in \mathbb{N}$, the maximum time it takes to serve a total of $\sigma \in \mathbb{N}$ service units in a busy time, as $\omega^+(\sigma) = \inf\{t_\delta \in \mathbb{N} \mid \xi(t_\delta) \geq \sigma\}$. We now update Equation (6.1) to bound the k^{th} request of an arbitrary busy time. Let $\bar{\alpha}(k) \in \mathbb{R}^n_{\max}$ denote the symbolic arrival time of the k^{th} request. A bound $\bar{\phi}(k) \in \mathbb{R}^n_{\max}$ to the completion time of the request is given by Equation (6.8), where $\bar{\alpha}(1)$, the arrival time of the first request, is the start of the busy time.

$$\bar{\phi}(k) = \bar{\alpha}(1) + \omega^+(\sigma(k)) \tag{6.8}$$

We also update Algorithm 4 to keep track of the busy times of resources in the symbolic simulation. For each resource, we track the start of the current busy time, say the l^{th} busy time, with a symbolic time-stamp $\bar{\alpha}^l(1)$ and the total

service units requested until and including the k^{th} request, i.e. $\sigma^l(k)$. Then, if the arrival time of the k^{th} request is before the completion of the $(k-1)^{th}$ request, i.e. $\bar{\alpha}^l(k) \preceq \bar{\phi}^l(k-1)$, then the k^{th} request is guaranteed to be in the same busy time and its symbolic finishing time is given by Equation (6.8). On the other hand, if $\bar{\alpha}^l(k) \not\preceq \bar{\phi}^l(k-1)$, the $\leq$ comparison is undecidable (or it is not guaranteed). Hence, we take the conservative assumption and start a new busy time at $\bar{\alpha}^{l+1}(1) = \bar{\alpha}^l(k)$. Furthermore, we conservatively assume a busy time does not span over multiple iterations and hence, all identified busy times are contained within an iteration [1]. We next show the analysis using the running example of Section 6.3.

Example: The first request on processor p_2 (firing of actor z) starts a busy time at $\bar{\alpha}^1(1) = [-\infty, -\infty, 0, -\infty, -\infty, 0]$. Hence, its response time is 3, which is the same as its WCRT. Its finishing time is at $\bar{\phi}^1(1) = [-\infty, -\infty, 3, -\infty, -\infty, 3]$. The second request (firing of actor y) is enabled at $\bar{\alpha}^1(2) = \max(\bar{t}_b, \bar{t}_d, \bar{t}_{p_2}) = [-\infty; 0; 3; 0; -\infty; 0]$, as shown below.

$$\begin{aligned}
\bar{\alpha}^1(2) &= \max(\bar{t}_b, \bar{t}_d, \bar{t}_{p_2}) \\
&= \max([-\infty; 0; -\infty; 0; -\infty; -\infty], [-\infty; -\infty; 3; -\infty; -\infty; 3]) \\
&= [-\infty; 0; 3; 0; -\infty; 0] \\
&\not\preceq [-\infty, -\infty, 3, -\infty, -\infty, 3].
\end{aligned}$$

Hence, $\bar{\alpha}^1(2) \not\preceq [-\infty, -\infty, 3, -\infty, -\infty, 3]$. Since $\bar{\alpha}^1(2) \not\preceq \bar{\phi}^1(1)$, the second request is assumed to start a new busy time at $\bar{\alpha}^2(1) = \bar{\alpha}^1(2)$. The finishing time of y is then $\bar{\phi}^2(1) = \bar{\alpha}^2(1) + \omega^+(\chi(y)) = [-\infty, 7, 10, 7, -\infty, 10]$. The third request is the second firing of actor z. It is enabled at $\bar{\alpha}^2(2) = \bar{\phi}^2(1)$. Thus, it is guaranteed to arrive in the second busy time and finishes at $\bar{\phi}^2(2) = \bar{\alpha}^2(1) + 8$, improving its WCRT from 3 to 1. Then collecting the final tokens, we form a new matrix, given as

$$\begin{bmatrix} t'_a \\ t'_b \\ t'_c \\ t'_d \\ t'_{p1} \\ t'_{p2} \end{bmatrix} = \begin{bmatrix} 4 & 11 & 14 & 11 & 4 & 14 \\ 4 & 11 & 14 & 11 & 4 & 14 \\ -\infty & 7 & 10 & 7 & -\infty & 10 \\ -\infty & 8 & 11 & 8 & -\infty & 11 \\ 4 & 11 & 14 & 11 & 4 & 14 \\ -\infty & 8 & 11 & 8 & -\infty & 11 \end{bmatrix} \cdot \begin{bmatrix} t_a \\ t_b \\ t_c \\ t_d \\ t_{p1} \\ t_{p2} \end{bmatrix}. \tag{6.9}$$

Compared to Equation (6.6), the new matrix improves the worst-case throughput, which is the inverse of the eigenvalue, from 0.077 to 0.091. This is a gain of 18% in tighter WCT bound. Such gains may largely vary with the application graph and scheduler configurations, as shown in Section 6.6.

[1] Refer to the discussion on future works in Chapter 8 how this can be further improved.

6.5 WCRC Derivation: The Case of CCSP

This section demonstrates derivation of WCRCs for a real-time arbiter, namely Credit-Controlled Static Priority (CCSP) arbiter [6]. The derivation follows an analytical approach and discusses two WCRCs in Section 6.5.2: an existing linear WCRC, called *latency-rate*, and our improved piecewise linear WCRC, called *bi-rate*. Section 6.5.1 first recaps the arbitration scheme of CCSP.

6.5.1 Credit-Controlled Static Priority (CCSP)

CCSP [6] arbitrates a shared MPSoC resource between a set R of *requestors*. For the analysis, an abstract resource view is used where a *service unit* is the access granularity of the resource, which can be served in a time-unit of one *service cycle*. Requests arriving from each requestor to the resource are placed in a separate buffer in front of the resource. Each requestor has a unique priority level. The arbiter grants access to the highest priority requestor that has requests waiting but also has enough resource budget [6], as explained next.

The service requests of a requestor are captured by a *requested service curve*, w and service provided to a requestor by the *provided service curve*, w'. These curves are illustrated in Figure 6.9. On such curves, $w(t_1, t_2) = w(t_2 + 1) - w(t_1)$ denotes the difference in values between the endpoints of the closed interval $[t_1, t_2]$. E.g. $w'(t_1, t_2)$ denotes the total service provided to a requestor in the interval $[t_1, t_2]$.

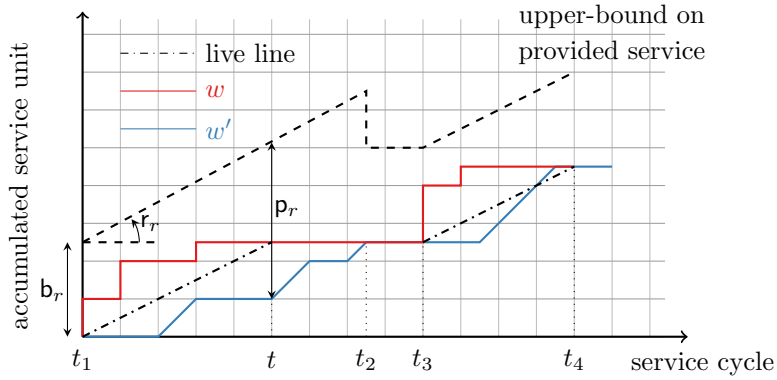


Figure 6.9: Allocated service, potential and active period.

The CCSP arbiter consists of a *rate regulator* and a *scheduler* [6]. The scheduler uses a static-priority scheduling scheme. Each requestor is assigned a unique priority level. The set of requestors that have higher priority than a requestor $r \in R$ is denoted R_r^+. The rate regulator is responsible for accounting the resource

budget of requestors. The amount of budget a requestor r has is referred to as its *potential* p_r. The potential of a requestor is determined by its allocated service, which consists of two parameters: the *allocated burstiness* ($\mathsf{b}_r \in \mathbb{R}$) and the *allocated rate* ($\mathsf{r}_r \in \mathbb{R}$). For a valid allocation, it should hold that $\sum_{r \in R} \mathsf{r}_r \leq 1$ and $\forall r \in R : \mathsf{b}_r \geq 1$. The potential is then defined as follows. Initially, the requestor has a potential equal to its allocated burstiness, b_r. Then, the requestor receives additional potential at its allocated rate, r_r, at every service cycle. Together these two parameters determine the *upper bound on the provided service* of the requestor, as illustrated in Figure 6.9. The potential of the requestor is decremented by one, for every service unit it receives.

This continuous budget replenishment policy is carried out for every *active period* of the requestor. The active period of a requestor is the maximum interval of time, during which (1) the requestor is backlogged, which means it still has requests waiting to be served, and/or (2) the requestor is *live*. A requestor is said to be live at time t if the total requested service is not less than the total service the requestor would receive if it were continuously getting service at its allocated rate; I.e. a requestor is live if $w_r(t_1 - 1, t_2 - 1) \geq \mathsf{r}_r \cdot (t_2 - t_1 + 1)$. A requestor in its active period interval is said to be active and R_t^a denotes the set of active requestors at time t. Figure 6.9 illustrates the relationships between allocated service ($\mathsf{r}_r, \mathsf{b}_r$), potential ($\mathsf{p}_r$) and active period ($[t_1, t_2]$ and $[t_3, t_4]$).

A requestor is said to be *eligible* for scheduling: (1) if it is backlogged and (2) if it has enough potential for at least one service unit i.e. $\mathsf{p}_r \geq 1 - \mathsf{r}_r$ (since the potential of an active requestor increments by r_r every service cycle). The set of eligible requestors at time t is denoted R_t^e. In summary, at every service cycle, CCSP grants access to the highest priority eligible requestor.

6.5.2 WCRC of CCSP

A WCRC specifies the minimum service a requestor is guaranteed to get over a time interval $[t_1, t_2]$. Thus, a WCRC $\check{w}'_r(t_1, t_2) \leq w'_r(t_1, t_2)$ is a lower bound on the provided service $w'_r(t_1, t_2)$. A WCRC is computed by considering the worst-case scenario that leads to the WCRT. For a requestor under the CCSP arbiter, this worst-case scenario happens when it experiences the maximum interference from higher priority requestors. According to [6], the maximum interference experienced by a requestor $r \in R$ during an interval $[t_1, t_2]$ occurs when all higher priority requestors start an active period at t_1 and remain active $\forall t \in [t_1, t_2]$.

It has been shown in [6] that a requestor is guaranteed to receive service at its allocated rate, r_r, after a maximum latency, $\mathsf{L}_r \in \mathbb{R}$. This WCRC, also known as the *latency-rate service guarantee*, ensures an active requestor $r \in R$ a minimum service during an active period $[t_1, t_2]$ according to Equation (6.10) and (6.11).

$$\forall t \in [t_1, t_2] : \ \check{w}'_r(t_1, t_2) = \max(0, \mathsf{r}_r \cdot (t_2 - t_1 + 1 - \mathsf{L}_r)) \tag{6.10}$$

$$\mathsf{L}_r = \frac{\sum_{s \in R_r^+} \mathsf{b}_s}{1 - \sum_{s \in R_r^+} \mathsf{r}_s} \tag{6.11}$$

The latency-rate WCRC guarantees that a requestor would receive service at its allocated rate, i.e. r_r service units every service cycle, after a maximum waiting latency of L service units. However, in CCSP a requestor can be temporarily served at a rate higher than its allocated rate after its maximum latency, which leads to a better service guarantee, as discussed next in Section 6.5.2.

Our Improved Bi-rate WCRC of CCSP

At the end of the maximum latency, a requestor can have an accumulated potential from two sources (according to the earlier discussion about potential): (1) from its allocated burstiness i.e. if $\mathsf{b}_r > 1$ and (2) from potential accumulated while being blocked by higher priority requestors, i.e. during the maximum latency, L_r. Thus, at the end of the maximum latency, a requestor has a potential that is equal to the sum of these two.

When a requestor is at the end of its maximum latency, it implies that higher priority requestors have utilized their accumulated potential. Thus, they have to accumulate potential at their respective allocated rate during multiple service cycles, before they are eligible to access the resource again. Consequently, the requestor can use the resource whenever it is not used by higher priority requestors. This means, the requestor can get service at a higher rate $\mathsf{r}_r^* \geq \mathsf{r}_r$, where

$$\mathsf{r}_r^* = 1 - \sum_{s \in R_r^+} \mathsf{r}_s. \tag{6.12}$$

The requestor receives service at this higher rate as long as its potential does not go below $1 - \mathsf{r}_r$, which is the minimum potential a requestor needs to have to be eligible for scheduling. The service cycle at which the potential drops below $1 - \mathsf{r}_r$ is referred as the *boundary cycle*, t_r^b. After the boundary cycle, the requestor has to wait multiple service cycles and accumulate potential at its allocated rate to be eligible. Therefore, it receives service at its allocated rate, r_r. The boundary cycle is given as $t_r^b = min(t_2, \lfloor t_x \rfloor)$ where

$$t_x = t_1 + \frac{\mathsf{b}_r - 1 + \mathsf{r}_r + \sum_{s \in R_r^+} \mathsf{b}_s}{\mathsf{r}_r^* - \mathsf{r}_r}. \tag{6.13}$$

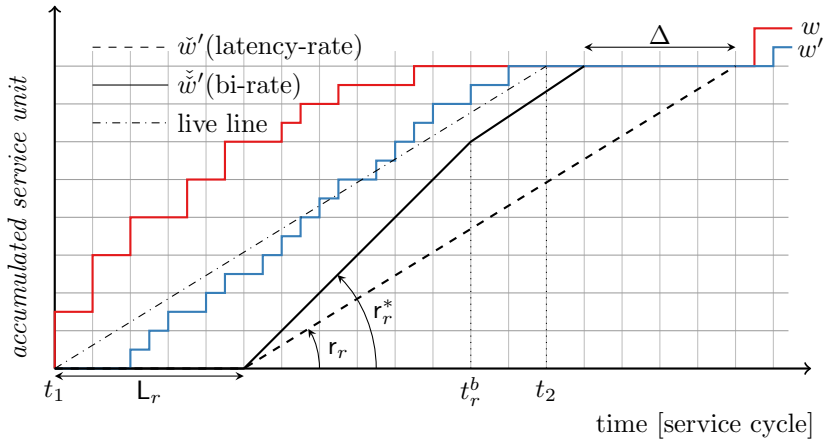


Figure 6.10: Improving WCRT of CCSP by a piecewise linear WCRC

The higher service rate results in a *bi-rate service guarantee* as the WCRC of CCSP. The bi-rate service guarantee improves the WCRT of requests compared to the latency-rate service guarantee, as shown in Figure 6.10. Δ in the figure illustrates the improvement in WCRT.

Figure 6.10 shows that under the bi-rate service guarantee, a requestor r is guaranteed a minimum service at two different rates, r_r^* and r_r, after a maximum latency L_r (given in Equation (6.11)). These two rates correspond to the cases when a requestor has enough potential to access the resource at a high rate, and when it does not. The bi-rate guarantee is defined based on two linear equations: higher-rate guarantee $\check{w}_r'^h$ and allocated-rate guarantee $\check{w}_r'^a$, given in Equations (6.14) and (6.15). The bi-rate service guarantee is, then, the lesser of the two, given in Equation (6.16). L_r^* is computed such that the intersection of the two linear equations, $\check{w}_r'^h(t_1, t_2)$ and $\check{w}_r'^a(t_1, t_2)$, is at t_x, given in Equation (6.13).

$$\check{w}_r'^h(t_1, t_2) = \mathsf{r}_r^* \cdot (t_2 - t_1 + 1 - \mathsf{L}_r) \tag{6.14}$$

$$\check{w}_r'^a(t_1, t_2) = \mathsf{r}_r \cdot (t_2 - t_1 + 1 - \mathsf{L}_r^*) \tag{6.15}$$

where

$$\mathsf{L}_r^* = -\frac{\mathsf{b}_r + \mathsf{r}_r^* - 1}{\mathsf{r}_r}$$

$$\check{\check{w}}_r'(t_1, t_2) = max(0, min(\check{w}_r'^h(t_1, t_2), \check{w}_r'^a(t_1, t_2))) \tag{6.16}$$

The bi-rate WCRC improves the WCRT of requests, compared to the latency-rate guarantee. This in turn leads to resource savings [75]. The mathematical proofs of the bi-rate WCRC derivation are also presented in [75].

6.6 Evaluation

We evaluate our SAAM mapping analysis approach in terms of analysis run-time and tightness of the computed temporal bounds. We have mapped different streaming applications onto a 4-tile platform model using the SDF3 [86] tool. The dataflow graphs are taken from the SDF3 [86] benchmark and from [59]. One of the dataflow graphs is shown in Figure 6.11, which is a sample-rate converter used in CDs. The application has only one scenario. The labels and WCETs of the actors are indicated inside the actors (e.g. $a, 5$).

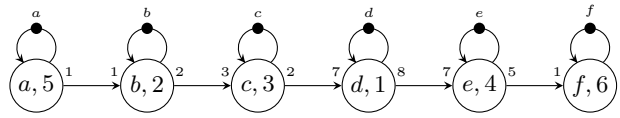


Figure 6.11: A sample-rate converter used in CDs

6.6.1 Analysis run-time

We compare our approach with the state-of-the-art RAD technique in the SDF3 tool [86]. For this purpose, we have also implemented our approach (SAAM) in the same tool. To analyse an application mapping, SDF3 first constructs a RAD model of each scenario mapping, as shown in Section 6.1.1. Then, it analyses the RAD models with different techniques, such as state-space [35], maximum-cycle-mean (MCM) (on homogeneous SDF graphs) [21] and $(max, +)$ automaton (MPA) [30]. However, the first two are not applicable to dynamic streaming applications. Hence, the constructed RAD models are analysed with MPA [30] and the results are compared with our approach, SAAM.

Consider the following mapping for Figure 6.11: $\{a, c\}$, $\{b, d, e\}$ and $\{f\}$ are mapped on processor tiles p_1, p_2 and p_3. The TDM frame is 100 time-units and the allocated slice is 50% on all tiles. The RAD model of this mapping has 14955 initial tokens. This is mainly due to the long SO schedule of the graph, which has a repetition vector of $[147, 147, 98, 28, 32, 160]$, and hence, a total of 612 actor firings per iteration. Due to the large number of tokens, the MPA analysis of the RAD model terminated without a result due to an out of memory exception. When the RAD model is directly analysed by state-space throughput analysis [35], it took more than 30mins. When the state-space analysis is optimized by embedding the SO schedule during the state-space analysis, the analysis is completed in 1820msec. Our approach completed the analysis in 72msec by limiting the initial tokens to just 25, giving the same throughput as the state-space analysis [35], 4.99×10^{-5} iterations per time-unit.

Table 6.1: Improvements in analysis runtime

Application	**Initial Tokens**		**Run-time (msec)**		**Speed-up**
	RAD	SAAM	RAD	SAAM	**avg/max**
H263encoder	444/567	104/169	$7/12\times10^3$	124/352	72/95
Modem*	456	54/-	5652	13/-	434
H263decoder*	2386	32/44	826×10^3	8/16	103×10^3
WLAN	62/72	32/46	22/36	5/12	5/20
LTE	646/1124	56/80	$196/366\times10^3$	85/144	$3/6\times10^3$
TD-SCDMA	31/39	11/15	4/8	1/4	4/8
Modem	95/119	54/82	42/80	13/32	4/8
H263decoder	75/88	32/44	24/36	8/16	3/7
H263encoder	402/465	104/169	4480/7792	119/524	57/103
MP3	60/68	41/52	15/24	5/8	3/12
LTE	80/101	56/86	956/4436	113/856	10/22
Sample-rate	1640*	21/28	-	56/88	-
RVC-MPEG	4385*	384/659	-	$22/88\times10^3$	-
Satellite	11642*	31/47	-	637/1132	-

The number of initial tokens of a RAD model changes with the mapping and the constructed SO schedule. Table 6.1 shows the average/maximum number of tokens and the corresponding run-time improvements. Hundreds of different mappings are analysed for each application. The table has three blocks, separated by double-lines. In the first block, all initial tokens are used in the analysis. In the second block, only manually selected tokens are used as an optimization step to reduce the problem size for RAD. The last block shows applications which terminated without results in RAD, with or without optimization, due to insufficient memory on a 2.4GHz dual-core PC with 4GB memory. (* entries are based on limited samples due to the long run-times of these mappings.)

Table 6.1 shows significant improvements. The gains translate to savings of hours during design-space exploration, since throughput is analysed repetitively. E.g. the reduction from an average of 4.4 to 0.12 seconds of H263encoder saves 11 hours for 10^4 mappings. Moreover, these gains are achieved without any loss of accuracy. In all cases, the computed throughputs of the two approaches are exactly the same if we do not use busy times in the analysis. We further gain in tighter bounds if busy times are used, as shown next.

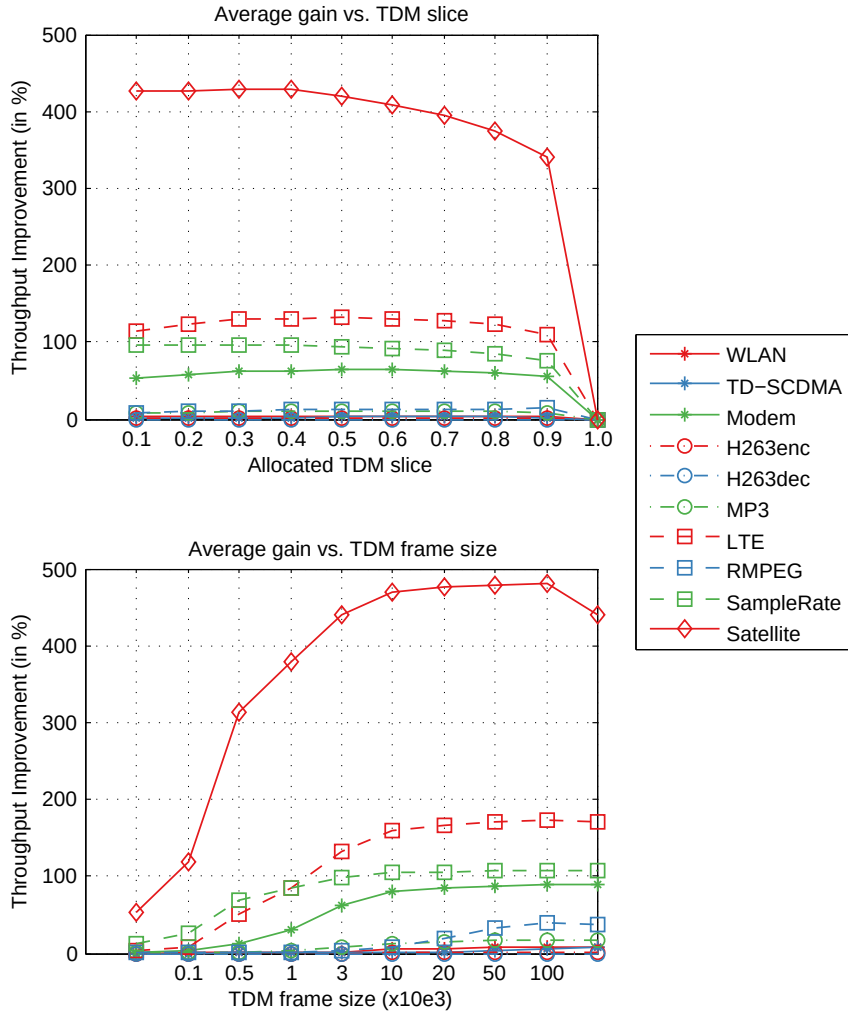


Figure 6.12: Improvements in worst-case temporal bounds

6.6.2 Tightness of performance bound

We have computed the worst-case throughput of each mapping in two ways for comparison: 1) using constant WCRT (Section 6.3) and 2) using multiple event busy times (Section 6.4). For the example mapping of Figure 6.11, given earlier in this section, the results are 4.99×10^{-5} and 8.80×10^{-5} iteration per time-unit, respectively. Thus, the latter improves the performance guarantee by 76% due to the improved WCRT analysis.

Figure 6.12 shows average improvements in throughput for different TDM frames and slice allocations. Applications with large repetition vectors, such as Satellite receiver, Sample-rate (SR) and LTE, have high gains, due to their long busy times. E.g. the Satellite receiver graph has 4515 actor firings per iteration, which gives it a long busy time. For a TDM slice allocation of 1, there is no gain, as expected, since the resource is fully allocated to the application and the worst-case waiting time is zero. The gain in the throughput bound improves as the TDM frame-size gets longer, since it increases the chance for multiple requests to fit within a slice of a frame, without extra waiting times due to preemptions. Generally, the result shows tighter performance bounds by using busy times in our symbolic analysis, without additional analysis run-time overhead.

6.7 Related Work

Compositional methods such as Real-Time Calculus (RTC) [17, 90] and SymTA/S [44, 73] apply concepts from network calculus to embedded streaming applications. They use event traffic propagation between resources to analyse the system. Their main strengths are modular analysis and support for priority-based as well as budget-based schedulers. Nevetheless, the aforementioned methods assume task graphs that are not sufficiently expressive to model adaptive applications, which change their graph structure, execution times and data rates dynamically. Even within a static mode, streaming applications require a task graph that supports cyclic-dependencies, multi-input tasks, multi-rate tasks (i.e. multiple tokens per input) and a natural way of handling back-pressured buffer communication. Our mapping analysis uses the FSM-SADF MoC, which allows modeling dynamic applications with variable port-rates and data-dependent workloads. As the same time, the MoC still allows us to verify basic properties such as deadlock-freedom and boundedness at design-time.

Existing dataflow-based mapping analysis approaches construct resource-aware dataflow (RAD) models [59,68,86,96,102] to analyse application mappings. These techniques account for resource allocations and scheduler configurations by introducing additional dataflow components into the application model to construct a RAD model. In the RAD model, the buffer-size of a channel is modeled by adding another channel in the reverse direction, with as many initial tokens as the allocated buffer-size [68]. Actors that are mapped on the same processor tile are executed following a static-order (SO) schedule, which can be modeled in a SDF scenario using the technique presented in [20]. Resources that are shared using arbiters under the class of latency-rate servers [84] can be modeled by the two-actors latency-rate dataflow model of [98].

RAD modeling of system-level mapping decisions has an advantage, since it maintains the same MoC as the application model. This implies that no new solutions are required to analyse mappings, as existing temporal analysis methods of the application MoC are immediately applicable. Yet, this approach has some drawbacks. First, dataflow components should be developed to model different types of resource arbitration schemes. This is not a trivial process. Up until now, only a limited number of models for predictable arbiters are developed [55,76,83,98]. The latency-rate model [98] partially addresses this challenge, as it offers a generalized linear model that is applicable to a number of schedulers under the class of $\mathcal{LR}$ servers [84]. However, the generalization comes at a price, as this simple linear model is not able to take unique features of individual arbiters into account for a tighter analysis. This may lead to pessimistic WCRT bounds, as demonstrated in [55, 76, 83]. A good example is the bi-rate WCRC of CCSP, presented in Section 6.5.2, which improves the latency-rate model and leads to resource savings [75]. To address the pessimism of simple general models, a detailed model, which is tailored to the arbiter under consideration, may be required. This brings us to the second challenge. Detailed models may introduce a large number of actors (which could be thousands for unfortunate combinations of parameters) into the application model. This can be seen in the TDM model of [55] and the SO model of [20]. As a result, detailed resource modeling may significantly dilate the graph, causing the analysis run-time to explode (cf. Section 6.1.1).

Our technique avoids constructing RAD models. Rather, it includes resources in the simulation, while embedding WCRCs in the dataflow analysis to characterize resource sharing. WCRCs are similar to lower-bound service curves of RTC [17]. We use such curves as flexible alternatives to represent a wide-range of resource arbitration strategies, while maintaining a good balance between accuracy and scalability. Unlike RTC, we do not compute through $(max, +)$ convolution the outgoing event stream and remaining capacity of a resource for a given input event stream (arrival curve). Rather, we use them to compute WCRTs of individual service requests, such as actor firings, for a duration of one graph iteration. This opens an opportunity to compute tighter WCRTs by identifying service requests, which arrive in the same busy time of the resource. The idea is known by the name *multiple event busy time* in the literature and has been used for tighter response time analysis [92] and improved event model construction [73]. In our work, we safely identify busy times of resources from the symbolic arrival times of requests, as presented in Section 6.4. Consequently, we avoid the assumption of having the critical instant on all service requests and improve WCRTs. It is also worth mentioning that our mapping analysis strategy can still be applied on a partial-RAD model if certain resources (such as DMA communication [25] and interconnects [42]) are found to be better represented by dataflow components.

6.8 Summary

This chapter has presented an efficient approach for temporal analysis of dataflow applications, mapped onto shared heterogeneous resources. The analysis approach avoids constructing resource-aware dataflow models, which are often used in existing approaches. It combines $(max, +)$-based symbolic simulation with worst-case resource availability curves. It avoids introducing tokens to model on-tile buffers as well as SO and budget schedulers. As a result, it keeps the graph size intact and improves scalability, which makes it tens of times faster than the state-of-the-art. Moreover, it gives tighter temporal bounds, up to a factor of 4, compared to the typical worst-case analysis, by improving the WCRTs of requests that arrive in the same busy time.

CHAPTER 7

Analysing Maximum End-to-end Latency

The throughput analysis of dataflow applications and their mappings have been presented in Chapter 5 and 6. Another important real-time property of streaming applications is end-to-end latency. In a wireless application, for instance, the acknowledgement of a properly received packet has to be sent within a certain timing duration, which is often set by the standard. Such kind of timing constraints imposes a real-time latency requirement that must be met. This chapter presents a design-time analytical technique to derive a conservative upper-bound to the maximum end-to-end latency of an application mapping. It formalizes the latency analysis problem in the presence of dynamically switching scenarios, modeled by the FSM-SADF MoC. Each scenario mapping is characterized by a $(max, +)$ matrix, as discussed in Section 6.3. The resulting matrices are then composed to derive a bound to the end-to-end latency under a periodic source. Aperiodic sources such as sporadic streams can be analyzed through reduction to a periodic reference. Moreover, the technique is illustrated with a trade-off analysis in resource reservation under a throughput constraint. The chapter is organized in eight sections. Section 7.1 and 7.2 highlight the latency analysis challenges and then outline our approach. Section 7.3 formalizes the latency definition of FSM-based SADF models. Section 7.4 and 7.5 present the proposed latency analysis technique for a periodic source and then discuss other extensions such as aperiodic sources. Section 7.6 evaluates the techniques. Section 7.7 discusses related work. Section 7.8 concludes the chapter.

7.1 Introduction

This chapter presents a design-time analytical approach to derive a conservative bound to the maximum end-to-end latency of a streaming application, while considering the different operating modes of the application. It assumes the application is modeled with the FSM-SADF MoC and is mapped on a heterogeneous MPSoC that is shared between multiple applications. It has been shown in previous chapters that FSM-SADF gives a tighter throughput guarantee than a static SDF model. This eventually leads to resource savings, since real-time requirements can be guaranteed at a lower resource allocation.

Nevertheless, the maximum end-to-end latency, which is a key real-time requirement, has never been studied in the presence of dynamically switching scenarios. For instance, in WLAN, whose FSM-SADF model is shown in Figure 3.6, an acknowledgment packet must be sent within $16\mu sec$ of the reception of the last OFDM symbol if the cyclic redundancy check (CRC) is successful. This time guard of $16\mu sec$, known as the Short Intra-Frame Spacing (SIFS), is a latency constraint that must be satisfied [59]. A WLAN packet may have a maximum of 256 OFDM symbols. Payload decoding is executed by scenario s_3 of Figure 3.6. Thus, s_3 may be executed multiple times before the CRC is executed by s_4. This implies that the maximum timing distance between the last execution of s_3 and the execution of s_4 must be computed to verify if the SIFS latency constraint is met. Cellular connectivity standards also have radio-trip time (RTT) requirements (e.g. $10msec$ for LTE), which specifies the maximum communication delay from a terminal to the network and back to the terminal. Such RTT requirements leave only a limited latency budget for baseband processing at the terminal [45].

The latency analysis of FSM-SADF has its own peculiarities and challenges, which deserve appropriate formalization and analysis techniques. For instance, existing latency analysis techniques in SDF are limited to computing temporal distance between events within the same iteration [36, 60]. However, analyzing latency in FSM-SADF requires computing the temporal distance that separates two causally-related events that are possibly multiple scenarios/iterations apart. This has two implications. First, the analysis should consider the overlapped transitions between scenarios to enable tighter bounds. Second, a bound to the maximum latency may not exist for an arbitrary FSM, since bounding the maximum latency also requires bounded-length scenario sequences. If the number of scenario sequences between the two causally-related events is unbounded (i.e. if there are cycles in between), the latency becomes unbounded. Thus, it is crucial to identify FSMs, whose end-to-end latency can be bounded. This chapter introduces a *latency automaton* that defines the class of FSMs with bounded-length scenario sequences between source and sink scenarios.

7.2 Approach

The main contribution of the chapter is a systematic approach to analyze the end-to-end latency of a dynamic streaming application, which has a set of dynamically switching scenarios modeled with an FSM-SADF. Latency is formally defined in terms of two causally related actor firings between a source and a sink actor, in a source and a sink scenario, respectively. The FSM can possibly have multiple source and sink scenarios. All possible scenario sequences between source and sink scenarios are then extracted from the FSM. Each scenario sequence is characterized by a matrix in $(max, +)$ algebra [43] that captures its end-to-end timing behavior. A bound to the maximum latency is then derived in two different ways with respect to a periodic source: 1) from a state-space, constructed in a breadth-first-search manner, by considering the possible transitions between scenario sequences, and 2) from a spectral analysis of the single-matrix characterization of all scenario sequences. Both techniques give the exact maximum latency if any arbitrary transition is possible between the scenario sequences. Otherwise, when specific sequences are specified by the FSM, the state-space technique still gives the exact maximum latency, while the single-matrix characterization only gives a conservative bound. The analysis of aperiodic sources, such as sporadic input streams, can be carried out through reduction to a periodic reference, by making use of the linearity property of $(max, +)$ systems.

Both analysis options have time complexity which is polynomial with the number of initial tokens of the graph. Thus, reducing the number of tokens becomes crucial for a faster analysis. To that end, we analyse scenario mappings without constructing resource-aware dataflow (RAD) models, which are often used in existing dataflow-based analysis techniques [20, 59, 86]. RAD models are generated by modeling scheduling and resource allocation decisions into the application model. Such additions may significantly dilate the RAD model and result in poor scalability as the graph size grows. We instead use a symbolic execution of scenario mappings that result in compact $(max, +)$ matrices. The construction of such compact matrices is presented in detail in Chapter 6.

We demonstrate the latency analysis technique with dataflow models from the wireless application domain. Moreover, we present a trade-off analysis in resource reservation under a throughput constraint. The throughput constraint is set by the source's period, which is modeled by the WCET of a source actor. Our analysis detects if the throughput constraint is not met, which implies that the end-to-end latency is unbounded. In this case, the analysis gives the minimum period the application supports. The evaluation shows that the approach has a low run-time that enables it to be effectively integrated in multiprocessor design flows for streaming applications.

7.3 Problem Formulation

A streaming application processes a sequence of input data objects (packets, frames, etc). It then produces the corresponding sequence of output data objects. We define a *stream* as a sequence of arrival/production time-stamps of these data objects, as defined in Definition 17.

Definition 17 (Stream)**.** *A stream $\bar{\varphi}$ is an element of the set $\mathbb{R}_{\max}^{\mathbb{N}}$ such that for $\bar{\varphi} = \langle \bar{\varphi}_1, \bar{\varphi}_2, \cdots, \bar{\varphi}_n, \cdots \rangle$ and $n \in \mathbb{N}$, $\bar{\varphi}_n \in \mathbb{R}_{\max}$ is the time-stamp of the n^{th} element of the stream.*

The time-stamp of the n^{th} element of a stream denotes the arrival time of the n^{th} input or the production time of the n^{th} output. As further discussed in Section 7.3.1, we construct streams in such away that there exists a one-to-one causal relation between the elements of an input stream and its corresponding output stream. Given an input stream and its corresponding output stream, the maximum end-to-end latency is determined by the longest timing separation between their elements, as defined in Definition 18.

Definition 18 (Maximum latency)**.** *Given an input stream $\bar{\varphi}_{in} \in \mathbb{R}_{\max}^{\mathbb{N}}$, its corresponding output stream $\bar{\varphi}_{out} \in \mathbb{R}_{\max}^{\mathbb{N}}$ and a dependency distance $d \in \mathbb{N}$ between corresponding input-output elements, the maximum latency $l \in \overline{\mathbb{R}}_{\max}$ is defined as*

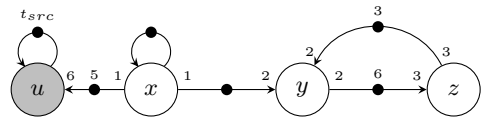
$$l = \max_{i \in \mathbb{N}} \big(\bar{\varphi}_{out}(i+d) - \bar{\varphi}_{in}(i) \big) \tag{7.1}$$

where $\bar{\varphi}_{out}(i+d)$ is the corresponding output element of input $\bar{\varphi}_{in}(i)$.

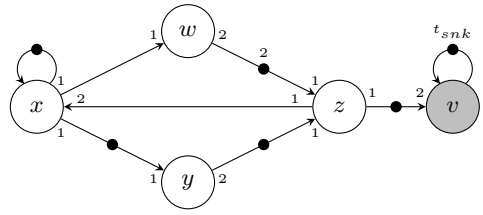
In Definition 18, the dependency distance $d \in \mathbb{N}$ denotes a delay where d output elements are produced even before the first input arrives, due to the initial state of the application. This section discusses the construction of input and output streams from scenario graphs and the identification of the causal relations between elements of the streams, along with their dependency distance.

7.3.1 Causality between Input and Output Streams

Given an application mapping (S, f, μ), the input stream is constructed from the firings of a single *source actor* a_{src} and the output stream is constructed from the firings of a single *sink actor* a_{snk}. We refer to a scenario that a_{src} belongs to as a source scenario $s_{src} \in S$. Similarly, a scenario to which the sink actor belongs to is referred to as a sink scenario $s_{snk} \in S$. There can be multiple source and sink scenarios, since source and sink actors may be active in multiple scenarios. A scenario can also be a source and a sink scenario at the same time.



(a) Source scenario. Actor u is the source actor.



(b) Sink scenario. Actor v is the sink actor.

Figure 7.1: Example source and sink scenarios. Actors u and v are added to record the time-stamps of input and output streams through tokens t_{src} and t_{snk}.

The firing times of the source actor form the input stream $\bar{\varphi}_{in}$. The firing times of the sink actor form the output stream $\bar{\varphi}_{out}$. We require the source and sink actors to fire exactly once per iteration to simplify the identification of causally related source and sink actor firings. Hence, the repetition factors of the source and sink actors should be 1 in the repetition vectors of their corresponding scenario graphs. Otherwise, new actors, which serve as source and sink actors, can be introduced with appropriate port-rates. The new actors ensure a repetition factor of 1, as shown in Figure 7.1. In the figure, actors u and v are introduced to record the firing times of actors x and z, respectively. The original source and sink actors (i.e. x and z) fire 6 and 2 times per iteration, respectively. The number of initial tokens on the channel c_{xu} determines which one of the 6 firings of actor x is recorded by the production time of token t_{src}. In the example, this is the first firing. A similar approach also works for channel c_{zv}. The newly added actors u and v become the source and sink actors, by replacing the original source and sink actors x and z. The time-stamps of token t_{src} constitute the input stream $\bar{\varphi}_{in}$. Similarly, the time-stamps of token t_{snk} constitute the output stream $\bar{\varphi}_{out}$.

Consider the execution of a sequence of scenarios $\langle s_{src}, s_1, s_2, \cdots, s_{snk} \rangle$ that begins with a source scenario and ends with a sink scenario and no other source or sink scenario in between. A firing of the source actor eventually influences some firing of the sink actor. However, the influence does not necessarily occur in a single execution of the sequence. This is due to initial tokens that may exist in the graph, which may cause the sink actor to fire, even before the source actor fires. We refer to the number of iterations of the sink scenario before a source

firing influences the sink actor as its *dependency distance*.

For instance, consider the graph of Figure 7.1a and assume it is both a source and a sink scenario, where actor x and z are the original source and sink actors, respectively. In the execution of this scenario, actor z can fire twice and complete its firings of the iteration before actor x fires. We are interested in one of the two firings of actor z, which will be recorded by the time-stamp token t_{snk}. This results in a dependency distance of 1, where the firings of actor x influences actor z in the next iteration. If there are multiple scenarios between a source and a sink scenario, the computation of the dependency distance becomes more involved (cf. Section 7.5.2 for further discussion).

We perform latency analysis by considering scenario sequences allowed by the FSM. However, the FSM can potentially specify infinitely many and infinitely long scenario sequences, due to cycles in the FSM. Scenario sequences that indefinitely stay within cycles without reaching a sink scenario may have no practical relevance. The latency of such sequences is also unbounded. Section 7.3.2 discusses a *proper latency automaton* that suffices to guarantee a bounded latency.

7.3.2 Latency Automaton

A common property of FSMs of real-life streaming applications is a *recurrent state* $q_r \in Q$. Streams are processed in fragments of data such as packets and frames. Applications often start processing a data fragment at a particular defined state. Then, they go through a sequence of states to process the data fragment and, eventually, return back to the recurrent state to process the next data fragment. However, the sequence of states an application goes through may vary with the content, size or type of the data fragment to be processed.

We define a *state-sequence* as a finite length sequence $\langle q_r, q_1, q_2, \cdots, q_n \rangle$ of states that starts with the recurrent state q_r. Section 7.4 later presents the latency analysis of FSM-SADF models whose FSMs specify a finite number of state-sequences with a common recurrent state. This requirement is formally specified by defining an automaton of a proper FSM for latency analysis, as defined in Definition 19. The definition states that a proper FSM for the latency analysis has a recurrent state q_r such that for every state $q \in Q$, any path starting from q must lead to q_r (i.e. has no infinitely long path, without revisiting q_r).

Definition 19 (Latency Automaton)**.** *Given an application mapping (S, f, μ), a latency automaton of FSM $f = (Q, q_0, T, \epsilon)$ is a tuple $(f, Q_\downarrow)$ such that $Q_\downarrow \subseteq Q$ is a set of final states. We say $(f, Q_\downarrow)$ is proper for latency analysis if it recognizes the language $\mathcal{L} = \{q_0 T (q_r P)^* \in Q^* \mid T, P \in Q^* : |T|, |P| \leq m\}$, where $q_r \in Q$ is a recurrent state, Q^* is the set of strings over Q and $m \in \mathbb{N}^0$.*

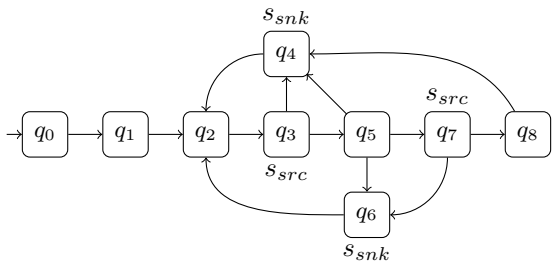


Figure 7.2: An example of a proper latency automaton

According to Definition 19, a proper latency automaton specifies transient sequences of states (denoted by q_0T), followed by repetitive concatenation of state-sequences (denoted by q_rP). In the sequel, we use the term state-sequence to refer to also transient sequences. Figure 7.2 shows an example of a proper latency automaton. It has six state-sequences, where q_2 is the recurrent state: $\langle q_0, q_1\rangle$, $\langle q_2, q_3, q_4\rangle$, $\langle q_2, q_3, q_5, q_4\rangle$, $\langle q_2, q_3, q_5, q_6\rangle$, $\langle q_2, q_3, q_5, q_7, q_6\rangle$ and $\langle q_2, q_3, q_5, q_7, q_8, q_4\rangle$. Among the six, q_0q_1 is the only transient sequence.

A state-sequence may contain source and sink scenarios in different possible orders. We define a *latency-sequence* recursively as a sub-sequence of a state-sequence such that the first state is labeled with a source scenario, the last state with a sink scenario and there is no other latency-sequence in between. The first and the last states of a latency-sequence can be conveniently referred to as source and sink states, respectively. Some examples of latency-sequences of Figure 7.2 are q_3q_4, $q_3q_5q_4$, $q_3q_5q_6$ and q_7q_6. The sequence q_3, q_5, q_7, q_8, q_4 is not a latency-sequence, since its sub-sequence q_7, q_8, q_4 is already a latency-sequence.

A state-sequence may have multiple latency-sequences. The latency-sequences in a state-sequence can be identified by associating or pairing each sink state with a source state. To do the pairing, we start from the last sink state and go in the reverse direction, as shown by the example of Figure 7.3. The last sink state is paired with the first source state that is encountered while going in the reverse direction. This gives the pair (q_9, q_6) in Figure 7.3, which defines the latency-sequence $q_6q_7q_8q_9$. By moving further in the reverse direction, the next sink state is paired with the first source state, which is not yet paired with another sink state. This gives the pair (q_8, q_4), which defines the latency-sequence $q_4q_5q_6q_7q_8$. If the number of source and sink states are properly matched, none of the source and sink states will be left unpaired at the end. Otherwise, some source or sink states will be left unpaired (in front of the other source-sink pairs). The implication of such unpaired source/sink states in the latency analysis is discussed in Section 7.5.2.

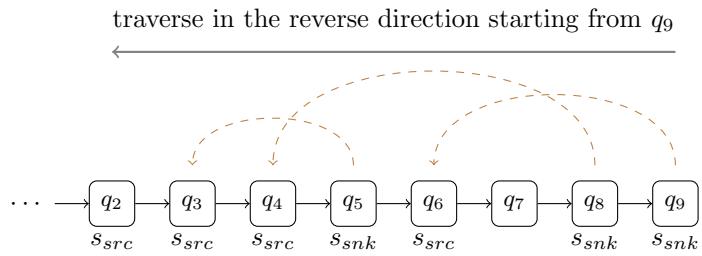


Figure 7.3: Identifying latency-sequences in a state-sequence

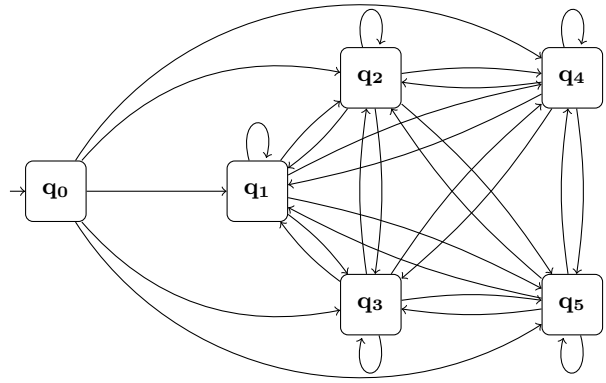


Figure 7.4: A condensed FSM of Figure 7.3

7.3.3 Condensed FSM

A proper latency automaton has a *condensed FSM* representation, which is constructed by replacing each state-sequence by a new FSM state. The condensed FSM comprises a finite number of transient states, followed by a finite set of fully connected states. Figure 7.4 shows the condensed FSM of the latency automaton of Figure 7.2, which has 6 state-sequences. Each of these state-sequences corresponds to a state in the condensed FSM of Figure 7.4, where $\mathbf{q_0} = \langle q_0, q_1 \rangle$, $\mathbf{q_1} = \langle q_2, q_3, q_4 \rangle$, $\mathbf{q_2} = \langle q_2, q_3, q_5, q_4 \rangle$, $\mathbf{q_3} = \langle q_2, q_3, q_5, q_6 \rangle$, $\mathbf{q_4} = \langle q_2, q_3, q_5, q_7, q_6 \rangle$ and $\mathbf{q_5} = \langle q_2, q_3, q_5, q_7, q_8, q_4 \rangle$. $\mathbf{q_0}$ is a transient state. The other states are fully connected, which implies that the state-sequences may occur in any arbitrary order. (As a post-optimization step, the transitions in the condensed FSM can be made to capture only the possible orders of executions of state-sequences).

Algorithm 5 Construct the states of a condensed FSM

```
1: CondensedFSMStates (f = (Q, q_0, T, ϵ), q_r)
2: Q ← the set of state-sequences
3: for every outgoing transition t = (q_r, q_s) of q_r do
4:    q_r := new state-sequence
5:    push back q_r to q_r
6:    findStateSequences(Q, q_r, q_r, t)
7: end for
8: return Q
```

Algorithm 6 Find state-sequences from a given transition t and add them $\mathbf{Q}$

```
1: findStateSequences (Q, q, q_r, t)
2: q_s := destination state of tranition t of FSM f
3: if q_s equals q_r then
4:    Q := Q ∪ {q}
5:    return terminate path          //state-sequence found
6: end if
7: for every q' in q do
8:    if q' equals q_s then
9:       return search fail          //unbounded loop detected
10:   end if
11: end for
12: for every outgoing transition t = (q_s, q_u) of q_s do
13:   q_s := new state-sequence
14:   copy q into q_s
15:   push back q_s at the end of q_s
16:   findStateSequences(Q, q_s, q_r, t)
17: end for
```

Algorithms 5 and 6 outline the construction of the states of a condensed FSM from a given proper latency automaton. The algorithms do not consider transient states for readability reasons. All transient states can be simply found from paths that start with the initial state of the FSM and end at the recurrent state q_r. Algorithm 5 starts from the recurrent state q_r and searches for state-sequences in every outgoing transition t (line 3). Each outgoing transition potentially leads to a new state-sequence. Hence, a new state-sequence is started for each outgoing transition (line 4) that begins with q_r (line 5). Then, the search for valid state-sequences continues from the destination state of transition t (line 6) using the recursive procedure of Algorithm 6.

Algorithm 6 recursively searches for state-sequences starting from a given transition t with destination state q_s (line 2). Every outgoing transition of state q_s (line 12) potentially starts a new state-sequence (line 13). The new state-sequence basically contains all sequences that led upto, and including, state q_s (line 14) and (line 15). Then, the search continues from each of the immediately reachable states of q_s (line 16). A state-sequence is found if a path reaches a recurrent state q_r (line 3-6). A path may also return to an already traversed state before reaching the recurrent state (line 8). This indicates a cycle that creates an infinitely long (or unbounded) state-sequence. In this case, the algorithm terminates unsuccessfully (line 9), claiming the input FSM is not a proper automaton for the latency analysis[1]

If the algorithm succeeds, it returns a set of state-sequences. Each state-sequence forms a state of the condensed FSM. Each state of the condensed FSM is annotated with a matrix, which is the matrix product of the state-sequence matrices. A state-sequence $\langle q_r, q_1, q_2, \cdots, q_n \rangle$ has a corresponding scenario sequence $\langle s_r, s_1, s_2, \cdots, s_n \rangle$, where $s_x = \epsilon(q_x)$ is the scenario associated with state q_x, according to Definition 4. The execution of this scenario sequence from a time-stamp vector $\bar{\gamma}_{k-1}$ yields a new vector $\bar{\gamma}_k = M_n \cdot M_{n-1} \cdots M_1 \cdot M_r \cdot \bar{\gamma}_{k-1}$. Matrix M_x is the matrix of the scenario mapping of s_x, which is constructed using the approach presented in Chapter 6. Due to associativity of $(max, +)$ matrix multiplication, $\bar{\gamma}_k = M \cdot \bar{\gamma}_{k-1}$, where M is given as $M = M_n \cdot M_{n-1} \cdots M_1 \cdot M_r$. This way, each state of the condensed FSM gets a matrix that relates the end-to-end timing behavior of its scenario sequence. Once a condensed FSM is constructed and its states are annotated with matrices, a bound to the maximum latency is derived following the techniques presented in Section 7.4.

[1]To limit the occurrences of such cases, one improvement over the algorithms would be to start with a compact FSM, which has counters over its transitions. The compact FSM is then unfolded during the state-sequence construction. In this case, one may assign bounds to the transitions of the compact FSM. A bound of a transition specifies the maximum number of times the transition can be traversed while constructing a state-sequence. The algorithms can then keep track of the number of traversals of a transition by decrementing its bound. If a transition is not bounded, its bound can be assumed to be ∞ and the decrement operation does not affect it. If the transitions of a cycle of the FSM are bounded, the algorithm unrolls the cycle and constructs a bounded-length state-sequence. For instance, state q_2 of the FSM of WLAN, shown in Figure 3.6, can have a bound of 256. This bound exists because the maximum number of OFDM symbols in a WLAN packet is 256. Consequently, scenario $\epsilon(q_2) = s_3$, which performs the payload decoding, can be executed only for a maximum of 256 iterations per packet. The unrolling of the self-transition at state q_2 leads to 256 different state-sequences, in which the number of state q_2 varies from 1 to 256.

7.4 Analysing Latency under a Periodic Source

We compute latency between a given source actor and a sink actor. Any actor, except the source actor, can be a sink actor. However, the source actor is treated differently. A source is an external entity from an application's perspective. The arrival of input data objects from a given source, such as the RF frontend of a baseband modem, can be periodic or aperiodic, such as sporadic, bursty or periodic with jitter, as per the event model characterization suggested in [70]. This section derives a bound to the maximum latency under a periodic source. The analysis of aperiodic sources is handled by reduction to a periodic reference source using the approach of [60], as discussed in Section 7.5.1. Furthermore, in this section, the latency analysis is presented for the *basic case*. The basic case assumes a dependency distance of 0 (cf. Section 7.3.1) and a maximum of one latency-sequence per state-sequence (cf. Section 7.3.2). The extension of the analysis for the general case, where these two assumptions do not hold, is discussed in Section 7.5.2.

A periodic source is characterized by a period $p \in \mathbb{N}$ between consecutive firings. In dataflow graphs, a periodic source can be modeled by an SDF actor, with a self-edge, as shown by actor x of Figure 7.1. The period of the source is the WCET of the actor; i.e. $p = \chi(x)$. Next, we present two approaches to derive a bound to the maximum latency under a periodic source: *state-space analysis* and *spectral analysis*. The former always gives the exact maximum latency. The latter gives only a conservative bound if the FSM is not fully connected. In general, the latter has a lower run-time, specially in case of large number of state-sequences, as shown later in Section 7.6.

7.4.1 State-Space Analysis

An execution of FSM-SADF is an execution of a sequence of states allowed by the condensed FSM. The execution of a state is given by the relation $\bar{\gamma}_k = M \cdot \bar{\gamma}_{k-1}$, where M is the $(max, +)$ matrix of the scenario sequence of the state, as discussed in Section 7.3.3. In the basic case, there is at most one latency-sequence per state. If a latency-sequence exists in a state, the source and sink actors fire exactly once during the execution of the scenario sequence of the state. The firing times of these two actors are then recorded by the time-stamp tokens t_{src} and t_{snk}, respectively. We define the latency of the scenario sequence, in isolation, as the relative timing distance between the time-stamps of the source and the sink time-stamp tokens, i.e. t_{src} and t_{snk}. These two time-stamps form the i^{th} input-output element pair, $\bar{\varphi}_{in}(i)$ and $\bar{\varphi}_{out}(i)$ (cf. Definition 18), since dependency distance $d = 0$ in the basic case. The latency of the scenario sequence is given by Equation (7.2). Note

that $[\bar{\gamma}]_t$ denotes the entry of token t in vector $\bar{\gamma}$ and $\bar{\bar{\gamma}}$ is a normalized vector.

$$\begin{aligned} l_i &= \bar{\varphi}_{out}(i) - \bar{\varphi}_{in}(i) \\ &= [\bar{\gamma}_k]_{t_{snk}} - [\bar{\gamma}_k]_{t_{src}} \\ &= [\bar{\bar{\gamma}}_k + \|\bar{\gamma}_k\|]_{t_{snk}} - [\bar{\bar{\gamma}}_k + \|\bar{\gamma}_k\|]_{t_{src}} \\ &= [\bar{\bar{\gamma}}_k]_{t_{snk}} - [\bar{\bar{\gamma}}_k]_{t_{src}} \end{aligned} \tag{7.2}$$

Equation (7.2) needs to be computed for all reachable time-stamp vectors to find the maximum latency. The set of all reachable time-stamp vectors can be found using *state-space exploration*. The state-space is constructed in a breadth-first-search manner from the condensed FSM, following the approach of [30]. States in the state-space consist of a state-vector pair $(\mathbf{q}, \bar{\gamma}_{k-1})$, where $\mathbf{q}$ is a state of the condensed FSM, executed from a time-stamp vector $\bar{\gamma}_{k-1}$. Executing $\mathbf{q}$ from vector $\bar{\gamma}_{k-1}$ yields a new vector $\bar{\gamma}_k$. Since the FSM is non-deterministic, there may be multiple outgoing transitions from state $\mathbf{q}$. This results in multiple state-vector pairs $(\mathbf{q}', \bar{\gamma}_k)$, where $\mathbf{q}'$ a directly reachable state from $\mathbf{q}$. Continuing the execution from each of these new pairs yields either new or already-visited pairs. If there are no more new pairs generated, the exploration is terminated and the set of reachable time-stamp vectors are finite. The maximum latency is then derived from the set S_γ of all reachable vectors $\bar{\gamma}_k$, as given by Equation (7.3).

$$l = \max_{\bar{\gamma}_k \in S_\gamma} ([\bar{\bar{\gamma}}_k]_{t_{snk}} - [\bar{\bar{\gamma}}_k]_{t_{src}}) \tag{7.3}$$

An important question is if the state-space of state-vector pairs is finite. A sufficient condition for the finiteness of the state-space is the self-timed boundedness of the matrices of the states of the condensed FSM. An argument for the sufficiency of this condition is given as follows. The number of states of the condensed FSM is finite. Hence, the finiteness of the state-space is determined by the boundedness of the normalized time-stamp vector $\bar{\bar{\gamma}}$. The normalized vector specifies the relative distance between the time-stamps of initial tokens. If such distances are bounded, then the entries of the time-stamp vector can only take values from a bounded range. As a result, there will be finite number of possible time-stamp vectors and, hence, the state-space is also finite. The relative distances between initial tokens are bounded, since the application is driven by the source actor, and as a result, no part of the graph can run independent of the source firing. If there are tokens that are not dependent on the source (during the execution of the scenario sequence of a state of the condensed FSM), these tokens must evolve as fast as the source time-stamp token; otherwise, the matrix of the state would not have been self-timed bounded. Consequently, the maximum relative distance between time-stamps of initial tokens is bounded due to the periodic source.

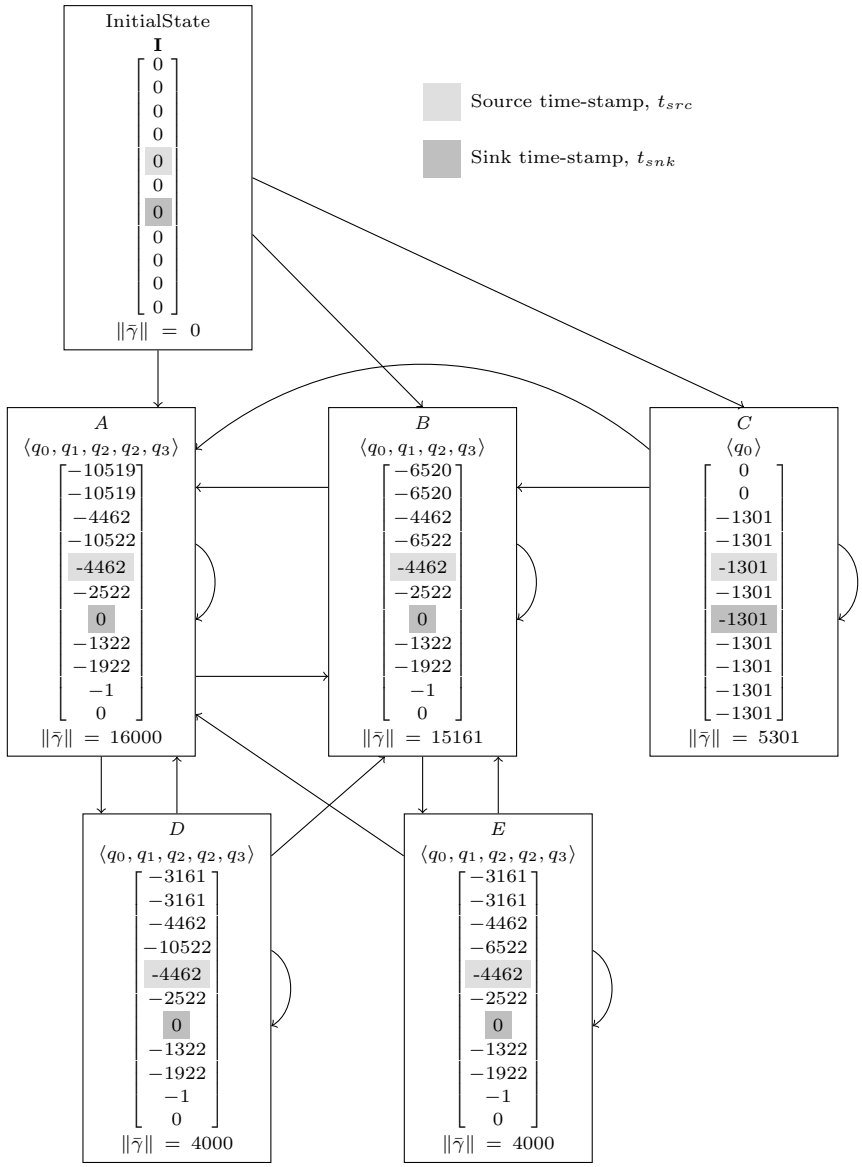


Figure 7.5: State-space of WLAN with a maximum payload of 2 symbols

If all matrices of the states of the condensed FSM are self-timed bounded, the state-space is finite and the latency computed by Equation (7.3) is the exact maximum latency. Otherwise, one may replace the unbounded matrices with conservative bounded matrices to obtain a finite state-space that gives a conservative bound to the maximum latency. If matrix M is replaced by a conservative matrix $\hat{M} \succeq M$, then $\hat{M}\bar{\gamma} \succeq M\bar{\gamma}$ from monotonicity (cf. Equations (7.6) and (7.7) to see how this is done). Hence, the replacement gives conservative time-stamps for the sink time-stamp token, while the time-stamps of the source token remains unchanged (as the firing of the source is periodic). This guarantees that the computed latency is a conservative bound to the maximum latency.

Figure 7.5 shows the state-space of WLAN (cf. Section 3.2) where the maximum payload size is limited to 2 OFDM symbols (to keep the state-space simple for illustration). This gives three state-sequences: $Sequence\ 1 = \langle q_0, q_1, q_2, q_2, q_3 \rangle$ for decoding a payload of two symbols, $Sequence\ 2 = \langle q_0, q_1, q_2, q_3 \rangle$ for decoding a payload of one symbol and $Sequence\ 3 = \langle q_0 \rangle$ for synchronization. The matrix of a state-sequence is obtained from the product of the matrices of the scenarios of the sequence, as discussed in Section 7.3.3. The matrices of Sequence 1, 2 and 3 are given in Equations (7.4),(7.5) and (7.6).

$$\begin{bmatrix}
1944 & 3244 & 9943 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
1944 & 3244 & 9943 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & 16000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
1941 & 3241 & 9940 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & 16000 & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
3883 & 5183 & 17940 & 1840 & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\
6405 & 7705 & 20462 & 4362 & -\infty & 2522 & 0 & 601 & 2522 & 1922 & 1 \\
5083 & 6383 & 19140 & 3040 & -\infty & 1200 & -\infty & 600 & 1200 & -\infty & -\infty \\
4483 & 5783 & 18540 & 2440 & -\infty & 600 & -\infty & -\infty & 600 & -\infty & -\infty \\
6404 & 7704 & 20461 & 4361 & -\infty & 2521 & -\infty & -\infty & 2521 & 1921 & -\infty \\
6405 & 7705 & 20462 & 4362 & -\infty & 2522 & -\infty & 601 & 2522 & 1922 & 1
\end{bmatrix} \tag{7.4}$$

$$\begin{bmatrix}
1943 & 3243 & 9942 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
1943 & 3243 & 9942 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & 12000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
1941 & 3241 & 9940 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & 12000 & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
3882 & 5182 & 13940 & 1840 & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\
6404 & 7704 & 16462 & 4362 & -\infty & 2522 & 0 & 601 & 2522 & 1922 & 1 \\
5082 & 6382 & 15140 & 3040 & -\infty & 1200 & -\infty & 600 & 1200 & -\infty & -\infty \\
4482 & 5782 & 14540 & 2440 & -\infty & 600 & -\infty & -\infty & 600 & -\infty & -\infty \\
6403 & 7703 & 16461 & 4361 & -\infty & 2521 & -\infty & -\infty & 2521 & 1921 & -\infty \\
6404 & 7704 & 16462 & 4362 & -\infty & 2522 & -\infty & 601 & 2522 & 1922 & 1
\end{bmatrix} \tag{7.5}$$

$$\begin{bmatrix} 1 & 1301 & 5301 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 1 & 1301 & 5301 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} \tag{7.6}$$

The matrices are based on eleven initial tokens from the FSM-SADF model of WLAN. The first seven initial tokens have the following indices in the matrix: $\{(a,1),(b,2),(c,3),(d,4),(t_{src},5),(e,6),(t_{snk},7),\cdots\}$. The other four initial tokens are from the self-edges of actors *crc*, *ack*, *spl* and *shr*. The matrices of Sequence 1 and 2 are self-timed bounded. However, the matrix of Sequence 3 is not. This is because tokens such as d, e, t_{src} and t_{snk} do not exist in Sequence 3. As a result, the matrix entries at (i,i) are set to zero for $4 \leq i \leq 11$, which makes the matrix self-timed unbounded. The zero entries on the diagonal imply that the time-stamps of these tokens do not change when Sequence 3 is executed.

$$\begin{bmatrix} 1 & 1301 & 5301 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 1 & 1301 & 5301 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 4000 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 4000 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 4000 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 4000 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 4000 \end{bmatrix} \tag{7.7}$$

To obtain a finite state-space, the matrix of Sequence 3 is replaced with a conservative self-timed bounded matrix $\hat{M} \succeq M$, shown in Equation (7.7). Matrix $\hat{M}$ is constructed by replacing diagonal entries (i,i) of M by $\|\eta\|$ if $[\eta]_i \leq \|\eta\|$, where η is the cycle-time vector (cf. Section 5.4) of matrix M. This technique is chosen so that the growth rate of the token associated with index i becomes $\|\eta\|$. As a result, matrix $\hat{M}$ gets a cycle-time vector whose entries are all the same, ensuring self-timed boundedness. The replacement enables to construct a finite state-space and derive a bound to the maximum latency.

The construction of the state-space, shown in Figure 7.5, begins with a zero vector at the initial state **I**. This initial state is required since any one of the three state-sequences can be the first to be executed. The initial state enables to execute all the three state-sequences starting from a zero vector. The execution results in three new time-stamp vectors, indicated in the figure as A, B and C. Three further executions from each of these three time-stamp vectors give two more new vectors D and E, besides returning to an already visited state. Additional executions do not lead to new vectors and, as a result, the state-space construction terminates. Sequence 1 and 2 have one latency-sequence, which is between states q_2 and q_3[2]. Sequence 3, however, does not have a latency-sequence. Thus, vectors A, B, D and E encode the timing distance between t_{snk} and t_{src}, which is the latency of the corresponding state-sequence. The maximum among these four relative distances, which is 4462, is a conservative bound to the maximum latency of WLAN when the number of symbols are limited to 2.

7.4.2 Spectral Analysis

The state-space approach gives the exact worst-case latency. However, it may suffer from state-space explosion [80], as the number of states increases. This section presents another analysis option that improves the analysis run-time. A faster analysis can be achieved by considering the fact that all states, except the transient states, are fully-connected in the condensed FSM. This is because the fully-connected part has a simple one-matrix representation [30]. This matrix is given as $M = \max_{\mathbf{q} \in \mathbf{Q}'} M_{\mathbf{q}}$ where $M_{\mathbf{q}}$ is the matrix of state $\mathbf{q}$ and $\mathbf{Q}'$ is the set of states, except the transient states. Once such a matrix is constructed, the latency analysis is carried out through the recurrence relation $\bar{\gamma}_k = M \cdot \bar{\gamma}_{k-1}$, where $\bar{\gamma}_0$ is the time-stamp vector after executing the transient state-sequences. The execution of the recurrence relation is terminated once the time-stamp vector returns to a recurrent vector i.e. $\bar{\bar{\gamma}}_k = \bar{\bar{\gamma}}_{k-n}$ for some $n \in \mathbb{N}$. This because the execution enters a periodic phase (cf. Section 5.2.3) and the time-stamp vectors will repeat themselves afterwards. The maximum end-to-end latency is then computed according to Equation (7.3), where S_γ is the set of all encountered time-stamp vectors. This technique improves the run-time of the latency analysis compared to the state-space alternative; in particular, when the number of state-sequences is large, as demonstrated later in Section 7.6.3. The improvement is achieved without any loss of accuracy as long as $\bar{\gamma}_0$ is the time-stamp vector at the end of the transient states (not a zero vector). A conservative analysis results if otherwise a fully-connected condensed FSM is assumed, including the transient states.

[2]The latency requirement of WLAN is further discussed in Section 7.6.1

7.5 Extensions

In Section 7.4, the latency analysis is presented for a periodic source. Furthermore, the basic case is assumed where dependency distance is 0 and there is at most one latency-sequence per state-sequence. This section discusses how the periodic source and the basic case solution can be extended for a more general case.

7.5.1 Aperiodic Sources

It has been shown in [60] that the latency analysis of aperiodic sources can be analysed by reduction to a periodic reference. In [60], two types of aperiodic sources are considered: *sporadic* and *bursty* sources. A sporadic source produces events non-deterministically at arbitrary moments, but with some minimum inter-arrival time of $d \in \mathbb{N}$. Figure 7.6 shows the events of a sporadic source. The corresponding events of a reference periodic source is also shown in dashed lines. The period of the periodic reference equals the minimum inter-arrival time d.

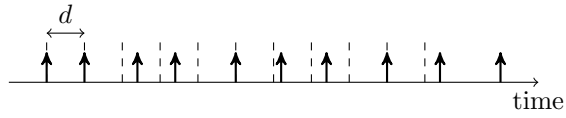


Figure 7.6: Sporadic events with minimum inter-arrival time of d

It has been shown in [60] that the maximum latency of a single-rate SDF graph under a sporadic source of minimum inter-arrival time d is the same as the maximum latency under a periodic source with period d. The basis of this result is the *linear timing* property of SDF graphs. Linear timing states that the production times of tokens cannot be delayed by more than the delays in the availability of input tokens [59, 96]. This can also be seen from the linearity property of $(max, +)$ matrices, as shown in Equation 7.8, where $\bar{\delta} \in \mathbb{R}^n_{\max}$ is a delay vector, whose entries all non-negative; i.e. $\bar{\delta} \succeq \mathbf{u}[0]$.

$$M \cdot (\bar{\gamma} + \bar{\delta}) \preceq M \cdot \bar{\gamma} + \|\bar{\delta}\| \tag{7.8}$$

It can also be seen from Equation 7.8 that FSM-SADF has linear timing property. This is because M can also denote the matrix of a sequence of scenarios. The linearity property enables to analyse the maximum latency of FSM-SADF under a sporadic source using a reference periodic source, by applying the same technique as [60]. The reference periodic source is modeled by a SDF actor, whose WCET equals the minimum inter-arrival time d of the sporadic source. Then, the maximum latency under a sporadic source is at most equal to maximum latency

computed for the reference periodic source. If an input from a sporadic source arrives later than its periodic reference by $\|\bar{\delta}\| \geq 0$ time units, the firing of the sink will be delayed in the worst-case by $\|\bar{\delta}\|$, due to linear timing property. Consequently, the relative maximum timing distance between the sink and the source time-stamp tokens does not increase; i.e. $(t_{snk} + \|\bar{\delta}\|) - (t_{src} + \|\bar{\delta}\|) \leq t_{snk} - t_{src}$.

Another type of aperiodic source is a bursty input stream. Following the definition of [70], a bursty event model (t_i, T_b, N_b) is characterized by a minimum inter-arrival time $t_i \in \mathbb{N}$ and the maximum number $N_b \in \mathbb{N}$ of events that may occur in a time window of $T_b \in \mathbb{N}$. In [60], the latency analysis of single-rate SDF graphs under a bursty input stream is discussed. The approach derives a conservative upper-bound to the maximum latency under a bursty source by first reducing the analysis to a reference periodic source, and then, accounting for the maximum deviation from the periodic reference. This approach can also be applied to FSM-SADF models, by using the technique of [60]. The period of the periodic reference is derived from the highest average firing rate of the bursty source, which equals $\frac{N_b}{T_b}$. The maximum deviation is found for the interval where all the N_b events arrive with a separation of the minimum inter-arrival time t_i .

7.5.2 Extending the Basic Case

In Section 7.4, the latency analysis has been presented for the basic case, where there is at most one latency-sequence per state-sequence, the dependency distance is zero and there are no unpaired source and sink scenarios in a state-sequence. This section discusses how the basic case of the latency analysis can be extended to a more general case.

- *Multiple latency-sequences per state-sequence*: If there are multiple latency-sequences in a state-sequence, the computed latency from Equation 7.2 would be based on only the last latency-sequence of the state-sequence and the other latency-sequences would not be part of the analysis. This is because, at the end of the state-sequence, the source and sink time-stamp tokens t_{src} and t_{snk} respectively record the last firings of the source and sink actors. This is due to the fact that the time-stamp of the previous firing is overwritten, every time the source (or the sink) actor fires. As a result, the computed latency, as per Equation 7.2, takes only the last pair of source and sink actor firings (or the last latency-sequence) into account.

  This may lead to an underestimation of the latency. One solution to handle the analysis of multiple latency-sequences in a state-sequence is to carry out the analysis in multiple steps, by considering only one latency-sequence at a time. For instance, Figure 7.3 shows an example of a state-sequence

that has three latency-sequences. In this case, the analysis is carried three times. First, only q_3 and q_5 are labeled as source and sink scenarios. In the second and third analyses, only q_4 and q_8 and q_6 and q_9 are labeled as source and sink scenarios, respectively. The maximum latency is then given by the maximum among these three possibilities.

- *Non-zero dependency distance*: A non-zero dependency distance implies that the sink actor can fire some d number of times even before the source actor fires, due to the initial state of the application. This means that only the $(i+d)^{th}$ output element $\bar{\varphi}_{out}(i+d)$ is causally related to the i^{th} input element $\bar{\varphi}_{in}(i)$. The first d firings of the sink actor do not have corresponding source firings. Hence, for these firings, latency cannot be defined and they are left out of the latency analysis.

  Consequently, the latency analysis can be reduced to the case where the dependency distance is zero. This is achieved by starting the analysis from the state of the application where the first d sink firings are completed. To realize this, each state-sequence needs to be executed while blocking (not firing) the source actor, until the sequence deadlocks. The resulting scenario graphs will have new distributions of initial tokens, where the first firing of the sink actor has a causal dependency with the first firing of the source. The analysis matrices for the latency analysis are then constructed from the new scenario graphs, which guarantee that output element $\bar{\varphi}_{out}(i)$ is causally related to input element $\bar{\varphi}_{in}(i)$.

- *Unpaired source and sink scenarios in a state-sequence*: In Section 7.3.2, identification of latency-sequences from a state-sequence has been discussed. The technique associates/pairs each sink scenario with a source scenario, by traversing a state-sequence in the reverse direction. It is possible that some source (or sink) scenarios can be left without being associated with sink (or source) scenarios. Such unpaired scenarios are always in front of the paired source-sink scenarios (latency-sequences) of the state-sequence.

  The analysis in Section 7.4 allows for some source scenarios to be left unpaired. A good example is the case of WLAN, where the source scenario s_3 is executed multiple times (as many times as the number of OFDM symbols in a packet) before the sink scenario s_4 is executed. The latency requirement, which is between the last s_3 and scenario s_4, is properly captured by the latency-sequence of a state-sequence that decodes a packet[3]. This holds generally because all unpaired source scenarios are in front of the latency-sequences of a state-sequence. As a result, the source time-stamp token

[3]The latency requirement of WLAN is further discussed in Section 7.6.1

t_{src} gets overwritten as these latency-sequences are executed later in the sequence and, hence, the timing distance between t_{src} and t_{snk} eventually depends only on the actual latency-sequences (i.e. the paired source-sink scenarios). However, it is not possible for some sink scenarios to be left unpaired, since this would imply a non-zero dependency distance, where a sink actor can fire before the source fires.

7.6 Evaluation

This section evaluates the presented latency analysis techniques. It also demonstrates applicability in MPSoC design-space exploration. The evaluation has three sections. Section 7.6.1 analyzes the end-to-end latencies of two wireless baseband applications: WLAN 802.11a and 3GPP's LTE. Section 7.6.2 binds these applications onto a multi-core platform model and discusses trade-offs between resource allocation and achievable temporal bounds. Section 7.6.3 evaluates the run-time of the analysis techniques.

7.6.1 Applications

This section discusses the maximum end-to-end latencies of two wireless baseband applications, before they are mapped onto a multi-core platform.

WLAN 802.11a baseband processing

The baseband processing of WLAN 802.11a is discussed in Section 3.2. The corresponding FSM-SADF model is shown in Figure 3.6. The payload size of a WLAN packet may vary from 1 to 256 OFDM symbols. Each symbol has a duration of $4\mu sec$. Packets in WLAN arrive sporadically. Hence, the maximum latency is equivalent to a periodic source of period $4\mu sec$, as discussed in Section 7.5.1. After these symbols are demodulated and decoded, a cyclic-redundancy check (CRC) is performed. If CRC is successful, an acknowledgment must be sent within $16\mu sec$ of the end of reception. This defines a latency requirement between the reception of the last symbol and the sending of the acknowledgement. Hence, the source scenario is s_3 i.e. $\epsilon(q_2^i)$ and the sink scenario is s_4 i.e. $\epsilon(q_3)$. The source and sink actors are actors *src* and *spd*. Note that the last source scenario at q_2^i is only paired with a sink scenario. The other source scenarios are not paired with a sink scenario. This is allowed by the analysis, as explained in Section 7.5.2.

Due to the variable number of OFDM symbols in a WLAN packet, scenario s_3 is executed as many times as the number of OFDM symbols. The FSM of Figure 3.6 conservatively approximates this behavior by allowing an arbitrary number

of payload symbols, as indicated by the self-transition in state q_2. As a result of this cycle on state q_2, Algorithm 5 fails to construct a valid condensed FSM. To address this issue, we carried out the analysis on the actual FSM, which has 256 FSM states labeled with scenario s_3, compared to the conservative version with a single state (i.e. q_2). The 256 states have the form q_2^i, where $0 \leq i \leq 255$. This results in a condensed FSM that has 257 state-sequences. One state-sequence is $\langle q_0 \rangle$, which captures the execution of scenario s_1 repeatedly until synchronization succeeds. The remaining 256 sequences have the form $\mathbf{q_i} = \langle q_0, q_1, q_2^0, q_2^1, \cdots, q_2^i, q_3 \rangle$, that specifies the scenario sequence to decode a packet of length $i + 1$ symbols.

The latency of the model has been analysed with both the state-space (Section 7.4.1) and the spectral analysis (Section 7.4.2) techniques. The constructed state-space has 514 states (state-vector pairs). The obtained maximum latency bound equals $4.46\mu sec$ and the analysis took $1200msec$. In fact, all visited states, which have a latency-sequence, have given the same latency bound as the maximum latency. The spectral analysis also gives the same latency bound in this case. This is expected because the condensed FSM does not have transient state-sequences and the state-sequences are fully-connected. Moreover, the analysis run-time of the latter approach is only $84msec$.

Long Term Evolution (LTE) baseband processing

LTE's downlink frame consists of 10 sub-frames, as discussed in Section 3.1. Each sub-frame is $1000\mu sec$ long and has 14 OFDM symbols. Hence, the source produces OFDM symbols periodically with a period of $1000\mu sec/14 = 71.4\mu sec$. The FSM-SADF model of LTE, shown in Figure 3.4, comprises five scenarios $s_1 - s_5$ [80]. Sub-frame processing always starts at scenario s_1 (i.e. source scenario) and terminates at s_3, s_4 or s_5 (i.e. sink scenarios). This results in three possible state-sequences, which are also latency-sequences. The state-sequences correspond to the three possible sub-frame types in LTE. Figure 7.7 shows the condensed FSM. In the condensed FSM, state **I** is introduced as an initial state, which is annotated with a zero matrix. This is because execution may initially start with any one of the three possible sub-frame types.

If a state-sequence is slower than the rate of the source, the graph cannot meet the required throughput. The temporal distance between the source and the sink also diverges indefinitely. As a result, the maximum latency becomes unbounded. The application is also termed as *unbounded*. A state-sequence is bounded if and only if its matrix has a cycle-time vector[4] whose entries are all the same. This indicates that all actors have the same execution rate.

All three state-sequences of this LTE model are unbounded for the given source

[4]The computation of cycle-time vectors is studied in Chapter 5.4.

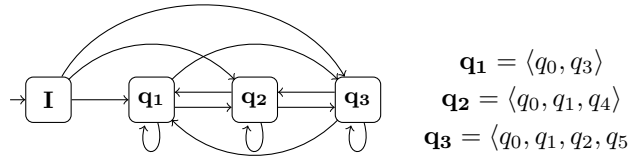


Figure 7.7: Condensed FSM of LTE, based on the FSM-SADF of Figure 3.4

period, which also means the application is too slow. This requires speeding-up critical actors so that the application can cope with the rate of the source. E.g. if actor *dec*, which performs data and channel decoding, is accelerated by a factor of 4, its execution time reduces and all state-sequences become bounded for the source period of $71.4\mu sec$. Thus, the acceleration of this actor makes the application bounded and it achieves the throughput constraint set by the source. The maximum latency is $1568\mu sec$, according to both the spectral and the state-space techniques. The results are the same since the condensed FSM is fully connected and has no transient sequences. The state-space has 13 states and the maximum latency is observed for state-sequence $\langle q_0, q_3 \rangle$. The analysis run-time is under $12msec$ for both analysis techniques.

If critical actors, such as actor *dec*, are not accelerated, the minimum period the graph can support is $127\mu sec$. Any period lower than this becomes too fast to be supported by the application. When the application is run under the minimum period, the maximum latency bound is $3752\mu sec$, according to both the state-space and the spectral techniques. The state-space has 157 states. The maximum latency is observed for state-sequence $\langle q_0, q_3 \rangle$ and the other sequences have a lower latency. The analysis run-time is under $20msec$ for both analysis techniques.

7.6.2 Resource Reservation vs. Temporal Bound Trade-offs

This section demonstrates how the presented latency analysis techniques can be used for a trade-off analysis between resource reservation and achievable temporal bounds for a software-defined radio (SDR) design using the same WLAN and LTE FSM-SADF models. A SDR comprises a heterogeneous MPSoC platform that is shared between multiple wireless radio applications. For instance, the baseband processor of high-feature phones is a MPSoC [19] that supports multiple applications, such as WLAN for wireless connectivity and WiMax and LTE for 3G/4G cellular connectivity.

Table 7.1: Constructed actor-to-processor bindings

Appl.	GPP	VP1	VP2	VP3	WPA1	WPA2	SNK
Binding-1							
WLAN	*mc, dda1, dda2, crc*	*shift, sync, hdem, pdem*	-	-	*hdec, pdec*	-	*ack, spld, shdr*
LTE	*mc,dda, mem, dem*	*est, adp, cqi, dmp*	-	-	-	*dec*	-
Binding-2							
WLAN	*mc, dda1, dda2, crc*	*shift, sync*	*hdem*	*pdem*	*hdec, pdec*	-	*ack, spld, shdr*
LTE	*mc,dda, mem, dem*	*est*	*adp, cqi*	*dmp*	-	*dec*	-

Platform

SDR platforms combine homogeneous and heterogeneous multiprocessing, similar to the platform in Figure 1.1. A typical SDR MPSoC comprises a general-purpose processor (GPP) (e.g. ARM) for control and generic tasks, a set of vector processors (VPs) (e.g. EVP [11]) for tasks such as synchronization and demodulation, and a set of weakly-programmable hardware accelerators (WPAs) for tasks such as Turbo decoding. Each processor tile has a local data and instruction memory, but has no cache. It is assumed that these processors are connected through a predictable interconnect [38] that uses a contention-free routing, based on TDM switching. Hence, each connection on the interconnect has a guaranteed bandwidth and maximum latency. For all connections, the same bounded delay WCRC (cf. Figure 6.4b) is assumed.

Application mapping

The application-to-platform mapping allocates resources such as buffer sizes of channels and TDM slices of applications. The design-space is vast for the system under consideration. Hence, we use fixed settings in this exercise for actor bindings and buffer sizes of channels to simplify the discussion. Channel bindings are infered from actor bindings. Hence, we explore the design-space with different TDM frame sizes and slice allocations, which give a trade-off between temporal bound and resource reservation.

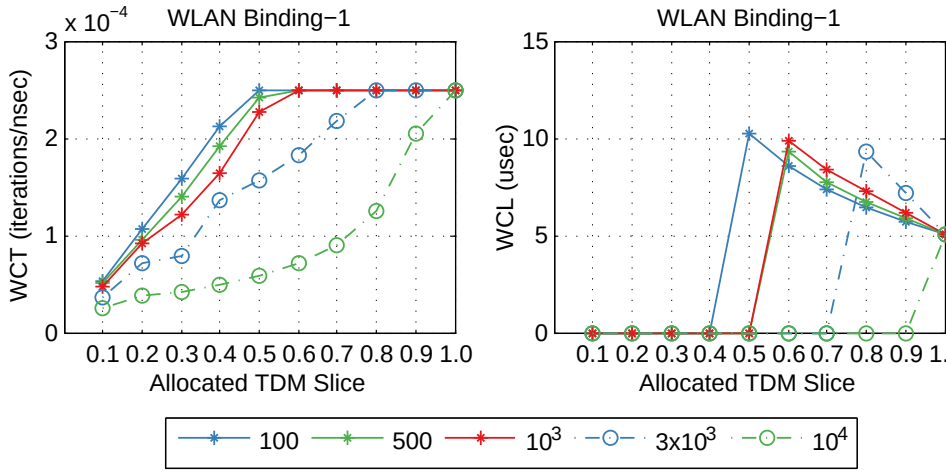


Figure 7.8: Timing Analysis of WLAN Mapping

We constructed two sets of actor-to-processor bindings, where control-oriented tasks are mapped on the GPP and signal processing tasks are mapped on the VPs, as shown in Table 7.1. A static-order schedule is constructed per tile between actors of the same application. The buffer-size of each channel c from actor p to actor q is set to a fixed value, equal to $2 \times u \times \nu(p) + \iota(c)$, where $\iota(c)$ is the number of initial tokens of channel c, u is the source port-rate of the channel and $\nu(p)$ is the repetition factor of source actor p. We set the buffer-sizes to fixed values in order to control the effects of variable buffer-sizes on the temporal analysis.

Both applications are intended to run together on the same platform, where TDM scheduling is used to arbitrate the processor tiles between them. Thus, TDM slices are allocated for each application, on each tile, on which the actors of the application are mapped.

Temporal Analysis

Figure 7.8 shows the worst-case throughput (WCT) and the worst-case latency (WCL) of WLAN Binding-1 for different TDM frame sizes and slice allocations. The results for Binding-2 are similar and are not shown here. The legend shows the different frame-sizes. The required throughput is 2.5×10^{-4} per $nsec$. The figure on the left shows that this throughput is not met until the slice allocation is sufficiently large (depending on the frame size). The figure on the right shows that a bound to the maximum latency does not exist when the WCT is less than the rate of the source (i.e. 2.5×10^{-4} per $nsec$). For instance, for a frame size of 100, all slice allocations less than 0.5 make the application unbounded. As a

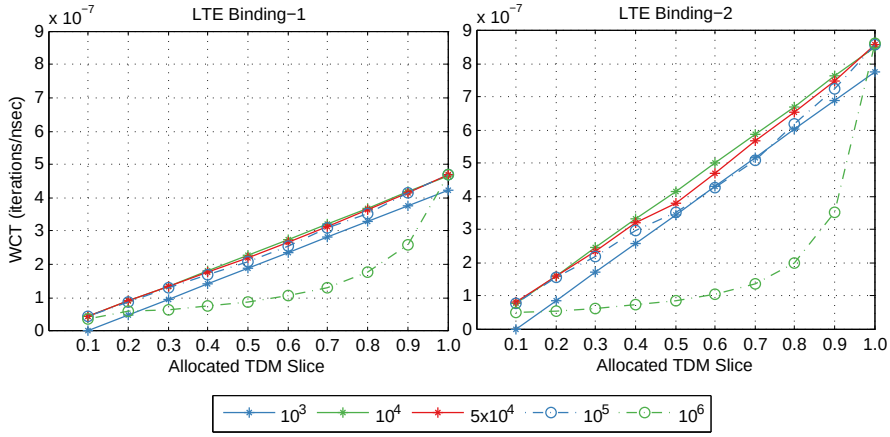


Figure 7.9: Timing Analysis of LTE Mapping

result, the maximum latency is also unbounded. Once the throughput requirement (which is the rate of the source) is met, increasing the resource reservation does not lead to a further increase in throughput. However, it helps to reduce the maximum end-to-end latency. The latency constraint is $16\mu sec$. The constraint specifies the maximum timing before sending an acknowledgement [59]. The latency analysis shown in Figure 7.8 demonstrates that all slice allocations that satisfy the throughput requirement satisfy this latency constraint.

Figure 7.9 and 7.10 show the timing analysis of the LTE processing in Binding-1 and Binding-2. Figure 7.9 shows that Binding-2 gives almost twice the throughput of Binding-1 for similar resource reservations. Figure 7.9 also shows that a TDM frame size of $10^4 nsec$ gives the best performance. Shorter frame sizes have lower performance due to extra overhead from frequent context switching. Longer frame sizes (e.g. $10^6 nsec$), on the other hand, have lower performance due to longer response times. However, this frame size of $10^4 nsec$ gives a degraded performance for WLAN if both applications are running together. As a compromise, one may select a frame size of $10^3 nsec$.

None of the resource reservations of the two LTE bindings satisfies the throughput requirement, which is dictated by the period of the source, $71.4\mu sec$. Thus, a bound to the maximum latency also does not exist for this source period (unless some critical actors are accelerated, as demonstrated in Section 7.6.1). The upper and lower figures of Figure 7.10 show the minimum supported period and the corresponding maximum latency bound, respectively, for different frame sizes and TDM slice allocations. For instance, for a TDM frame size of $10^3 nsec$ and allocated slice of 0.7, the minimum supported period equals $238\mu sec$ and a bound

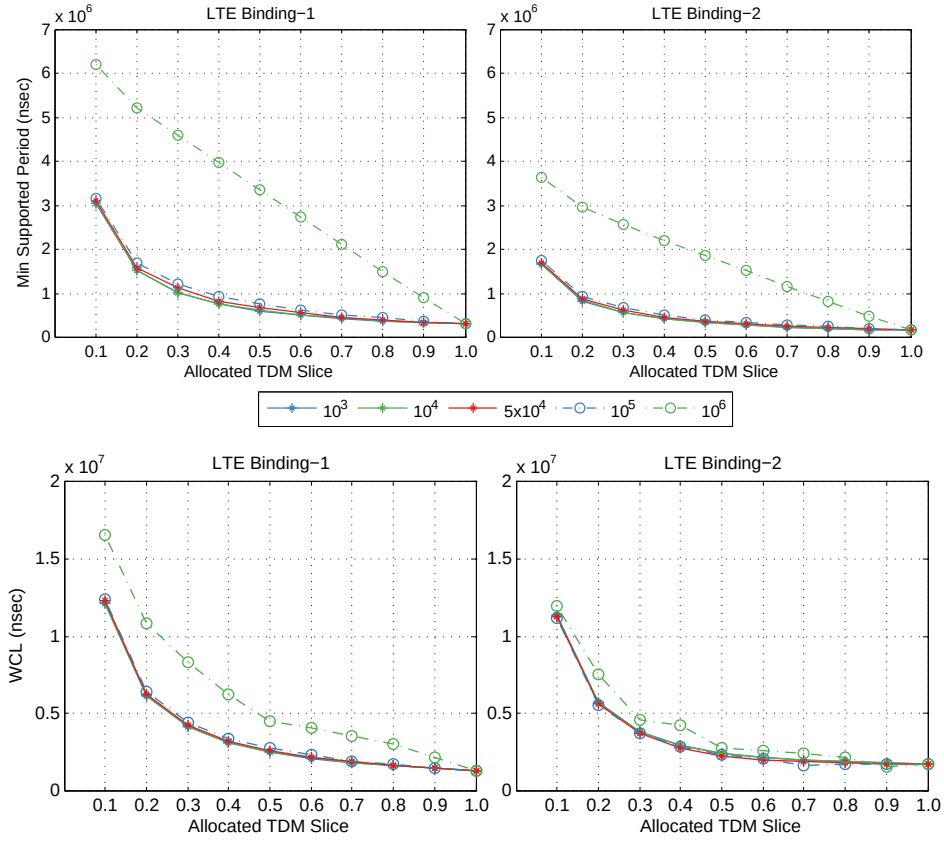


Figure 7.10: Timing Analysis of LTE Mapping

to the maximum end-to-end latency equals 1982μsec for binding-2.

7.6.3 Scalability

A key parameter that determines the run-time of the analysis techniques is the size of the matrices of state-sequences. Another parameter is the number of different state-sequences (or states) of the condensed FSM. Figure 7.11 shows how the run-time of the latency analysis scales with increasing number of initial tokens (and hence size of the matrices) and state-sequences. The experiment is conducted by randomly adding synthetic initial tokens and FSM transitions in the FSM-SADF model of LTE, shown in Figure 3.4. The run-times are shown for both the state-space (SS) analysis and the spectral analysis (SA) techniques.

The analysis with the SA technique involves three steps: 1) constructing the

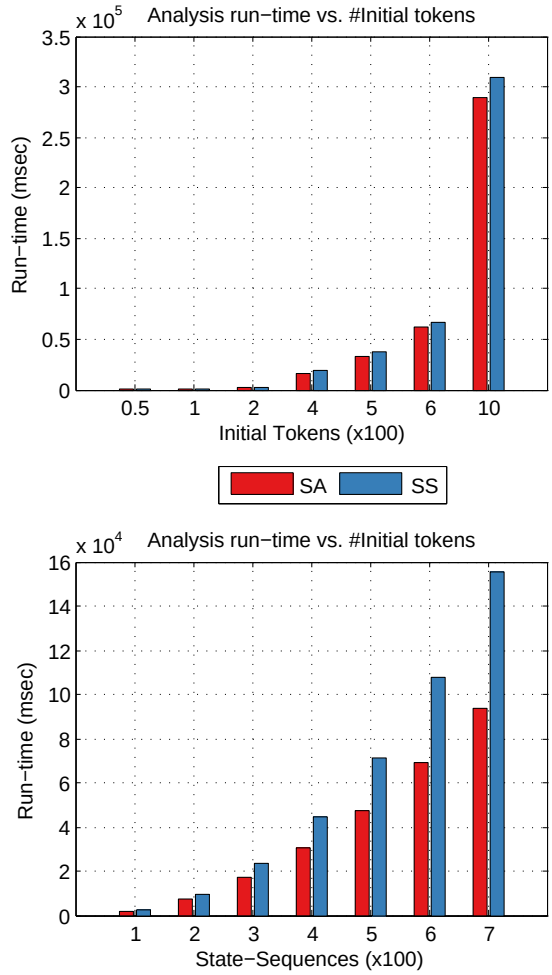


Figure 7.11: Analysis run-time for large problem sizes

matrices of the state-sequences, 2) a *max* operation on these matrices and 3) a spectral (eigenvalue) analysis of the final matrix. The time complexity of each of these three steps are discussed as follows. The matrices of state-sequences are of size $n \times n$, where n is the total number of initial tokens. This includes tokens that are used to capture processor and interconnect availabilities. Constructing the matrices (step 1) has a complexity of $\mathcal{O}(l \cdot m \cdot n^3)$, where l is the number of state-sequences, m is the maximum length of state-sequences and $\mathcal{O}(n^3)$ is complexity of matrix-to-matrix multiplication. The *max* operation (step 2) over these matrices is linear with the number of state-sequences and the size of the

matrices (n^2). The spectral analysis (step 3) uses Algorithm 3 of [28], which is based on the recurrent execution $\gamma_{k+1} = M \cdot \gamma_k$, until a periodic phase is reached. Thus, it has a complexity of $\mathcal{O}(t \cdot n^2)$, where t is the length of the transient phase and $\mathcal{O}(n^2)$ is complexity of matrix-to-vector multiplication. The length of the transient phase largely varies with the nature of the FSM-SADF graph, such as execution times and repetition factors of actors.

The analysis with the state-space technique also first requires constructing the matrices of the state-sequences (similar to step 1 of the spectral analysis). It further requires a matrix-to-vector computation of complexity $\mathcal{O}(s \cdot n^2)$, where s is the number of state-vector pairs of the state-space. The state-space is constructed through a breadth-first-search algorithm.

The size of the state-space may vary with the nature of the graph. For instance, it may increase exponentially with the number of state-sequences (states of the condensed FSM). This is because the execution of an FSM of l states starting from the initial state gives at most l and l^l new states, respectively, after the first and second rounds of executions. Consequently, the performance of the state-space technique may degrade with increasing number of states. This can also be seen from Figure 7.11, where the spectral analysis performs better than the state-space technique as the number of state-sequences increases (the bottom figure). However, the results show comparable performance between the two techniques when the run-time is evaluated for increasing number of initial tokens (the top figure). This is justifiable, as the number of state-sequences is low for this experiment (it has only three state-sequences); for the rest, both techniques have similar time complexity, which is polynomial with the number of initial tokens.

7.7 Related Work

Existing analytical techniques follow different approaches to compute exact or conservative upper-bounds to the maximum end-to-end latency of distributed real-time systems. These approaches differ in the expressiveness of the application model they use, the types of schedulers they support, the tightness of their analysis as well as their scalability and modularity [63].

Exploiting classical uni-processor scheduling to multi-core systems has been widely done to compute end-to-end latency in distributed real-time systems. These works can be categorized as modular or holistic. SymTA/S [44] is a modular approach where each system component is analyzed locally with classical algorithms. Then, the local results are propagated to other connected components through appropriate event model interfaces. The analysis is completed successfully if all event streams converge towards a fixed-point through an itera-

tive process. An end-to-end latency analysis based on the SymTA/S approach has been studied in [72]. The technique supports forks, joins and cyclic-dependencies in the task graph. However, it is limited to single-rate task graphs and does not support delays (also called initial tokens in dataflow MoCs). A related approach that integrates dataflow MoCs in the SymTA/S flow has also been presented in [74]. It allows a SymTA/S component to be modeled as a single-rate SDF graph, also called Homogeneous SDF (HSDF), which restricts all port-rates of the graph to be only 1. Holistic approaches [50, 67, 91] follow a different direction than SymTA/S to exploit classical uni-processor scheduling for multi-core real-time systems. They extend classical algorithms for specific combinations of task model, resource sharing and communication policy. As such, they integrate task and communication scheduling into a single analysis framework. Thus, they may give tight performance bounds by taking system-level correlations into account. However, these techniques are not modular and may require a completely new analysis for a new combination of schedulers and task model [63].

Another modular end-to-end latency analysis that supports arbitrary event models, so-called *arrival curves*, has been presented in [90]. The approach is based on Real-Time Calculus [17], which adopts concepts from network calculus to distributed real-time applications and shared resources. Real-time calculus employs *service curves* to model the availability of computation and communication resources. The approach propagates the output service curves of local resource analysis to connected components for a compositional performance analysis. By using service curves, the approach supports a wide range of scheduling policies. However, the application specification is still restricted to a single-rate task graph.

Dataflow MoCs have also been widely used to model and analyze streaming applications. In [60], maximum latency analysis for HSDF graphs has been studied under periodic, sporadic and bursty sources. The analysis derives a conservative latency bound after constructing a strictly periodic schedule (SPS) that gives upper-bounds to iterations. The analysis is first derived for a periodic source. Then, the analyses of sporadic and bursty sources are reduced to the analysis of a reference periodic source. The technique supports arbitrary cycles and delay tokens. However, analysis of an SDF graph requires conversion to an HSDF equivalent. In [36], a technique has been presented to compute the minimum achievable latency between a designated pair of SDF actors. The approach does not require conversion to HSDF and uses a state-space analysis, similar to [35], to find an execution scheme for the minimum latency.

The aforementioned and similar methods assume application models that are less suitable to model adaptive applications, which change their graph structure, execution times and data rates dynamically. Even within a static mode, streaming applications require a task graph that supports cyclic-dependencies, multi-input

tasks, multi-rate tasks (i.e. multiple data per input before a task is enabled), a natural way of handling back-pressured buffer communication and a (quasi-)static-order scheduling between tasks. The above approaches [50, 60, 72, 74, 90] support only single-rate tasks. Conversion from multi-rate to single-rate (HSDF) graphs is possible [81]. However, the conversion as well as the follow-up analysis may take tens of minutes for real-life application graphs, due to an exponential growth in the graph size [35]. Moreover, conservatively abstracting from the dynamism in modern-day applications leads to pessimistic temporal bounds, which may further result in unnecessary resource over-allocation [88].

Performance analysis techniques for FSM-SADF are presented in [28, 30, 79]. However, these works focus only on throughput analysis. This chapter studies the maximum end-to-end latency of a dynamic streaming application, which is modeled by an FSM-based SADF graph. The proposed analysis techniques in this chapter further consider the impact of resource sharing between multiple applications that are mapped on the same MPSoC. Unlike existing dataflow approaches [59, 86, 96], the presented latency analysis techniques do not require construction of resource-aware dataflow models to analyze resource sharing, as discussed in Chapter 6.

7.8 Summary

This chapter presents a systematic analytical approach to compute the exact or conservative maximum end-to-end latency of an FSM-SADF application model. The presented techniques determine a bound to the maximum latency with respect to a given periodic source. Aperiodic sources such as sporadic streams can be analyzed by reducing the analysis to the analysis of a periodic source. The analysis techniques have polynomial time complexity with the number of initial tokens. Thus, reducing the number tokens is crucial for a scalable analysis. The matrix construction algorithm of scenario mappings of Algorithm 4 helps to reduce the number of tokens, compared to resource-aware dataflow approaches. The technique is evaluated with existing dataflow models from the wireless applications domain. The evaluation also demonstrates the use of the analysis method for trade-off analysis in resource reservation under a throughput (source period) constraint. The technique can determine if the constraint is met. The technique can be combined with existing throughput and boundedness analysis methods to provide the minimum period the graph can support and the corresponding latency.

CHAPTER 8

Conclusion and Future Work

Current trends in embedded multimedia and wireless systems show that multiple applications are being integrated onto the same platform. Moreover, the applications are becoming computationally intensive, due to increasing demand for higher quality of service. Heterogeneous Multiprocessor System-on-Chip (MPSoC) has become the preferred choice of designers to implement such systems, as it provides the much-needed high performance at lower power consumption. An efficient implementation on such platforms requires an extensive design-space exploration (DSE), early in the design process. The DSE needs to find satisfactorily good design-points that meet real-time requirements at low resource cost. Dataflow models of computation (MoCs), as a means of achieving these goals, have been the focus of research in the past years. This thesis builds on the results of these researches, which have demonstrated the potential of dataflow-based approaches in designing predictable systems.

One aspect that plays a key role for the successful applicability of dataflow-based techniques is the sufficiency of expressiveness of the chosen MoC. Most importantly, the expressiveness should be sufficient enough to capture intra-application dynamism in modern-day streaming applications. The discussions in Chapter 3 have enlightened the types of dynamism in these applications. The chapter has demonstrated that these applications change their graph structures as well as their operating modes, depending on the processed input stream. Mode switchings occur dynamically, since different input data types may arrive in ar-

bitrary order. Along with the mode switchings, actors may dynamically change their input/output data rates, and even their functionalities. The chosen MoC in this thesis, FSM-SADF, has the expressiveness to capture such kinds of dynamism. It groups those actor properties that belong to the same mode into a SDF scenario. It allows scenarios to have different graph structures, actors to have variable WCETs and ports to have variable token rates. It further provides a FSM to encode the possible orders of executions of scenarios. Such kind of rich expressiveness, however, does not come for free. It brings new challenges when it comes to programmability and analysability. This thesis has made contributions towards addressing these challenges.

Chapter 4 introduces Disciplined Dataflow Network (DDN), as an analysable dynamic programing model, to tackle the programmability challenge. FSM-SADF has a *scenario-view* of the application, which is good for analysis, but not intuitive for programming. DDN has an *actor-view* of the application, where the application is represented by a network of dynamic actors. DDN provides a concise and programmer-intuitive representation of a dynamic streaming application. DDN is constructed in such a way that an FSM-SADF analysis model can be automatically extracted from it. As such, DDN serves as a common application specification for both the analysis and implementation trajectories. The basic idea is that the extracted model can be kept in the loop, performing different types of analysis and giving instant feedback to the designer. With the analysis framework, basic properties such as boundedness and deadlock-freedom can be checked. Deadlock-freedom is verified by analysing DDN actors individually as well as the extracted SDF scenarios. A SDF scenario is tested for deadlock-freedom by verifying consistency and the presence of sufficient initial tokens for one iteration [53]. In FSM-SADF, consistency is guaranteed if every cycle of the FSM is consistent [31]. Another property that can be checked is boundedness. Chapter 5 presents an algorithm to check boundedness of a scenario from the cycle-time vector of its matrix. Similar to consistency, boundedness of FSM-SADF is guaranteed if every cycle of the FSM is bounded.

The other challenge is temporal analysis. A designer requires to verify if his/her design choices would meet real-time constraints, before going to implementation. Decisions have to be made between multiple application partitioning options. After partitioning, actors and channels need to be bound to processors and the interconnect network, respectively. The result is an application mapping. The mapping also decides the resource share of the application if the platform is shared with other applications. The final set of design choices needs to be analysed to verify if the design can handle the arrival rate of the input source or can provide the sink with the required output data rate. Moreover, the analysis should be sufficiently fast so that the mapping and resource allocation design-

space can be efficiently explored iteratively to arrive at a low cost solution. In the light of these requirements, Chapter 6 has introduced an approach to analyse application mappings. The approach embeds worst-case resource curves (WCRCs) in $(max, +)$ symbolic simulation to model resource sharing. The benefit of this technique is that it avoids the construction of resource-aware dataflow models to analyse application mappings. The result is a more compact matrix representation of application mappings, which enables a faster timing analysis. Moreover, the approach improves the WCRTs of requests that arrive in the same busy time of a resource. This is achieved by identifying the busy times of an iteration from the symbolic time-stamps of actor firings. The technique enables tighter temporal bounds than existing approaches, which do not consider busy times in their analysis. The tightness of the temporal analysis is crucial, since the system dimensioning (e.g. buffer-allocation, processor budgets, etc) strongly depends on it. Otherwise, resources have to be unnecessarily over-allocated to be assured that real-time constraints are met.

An important real-time constraint in streaming applications is throughput. It determines the maximum input source rate the application can support, or the maximum output data rate it can provide to a sink. Such constraints come from customer requirements or standard specifications. Hence, they must be met. Chapter 5 has presented techniques to derive a bound to the worst-case throughput of SDF scenarios. The techniques are generalizations of existing methods in the literature. The generalizations become mandatory, since we intend to improve the expressiveness of SDF scenarios. Introducing new features to a MoC to improve its expressiveness, without supporting these features in the temporal analysis, will have no or little benefit in the design of predictable embedded systems. In the case of FSM-SADF, we have identified two important features that should be added to improve its expressiveness.

The first expressiveness feature we added is support for self-timed unbounded scenarios. In SDF, self-timed boundedness is crucial as it implies the number of tokens of a channel is bounded in a repeated self-timed execution of the graph [34]. In FSM-SADF, however, a SDF scenario is not necessarily repeatedly executed. Rather, a sequence of different scenarios is executed; one scenario followed by possibly another different scenario. Thus, it is sufficient to guarantee that such scenario sequences are bounded, irrespective of the boundedness of individual scenarios. The second required feature is to introduce a flexible way of encoding synchronizations between different scenarios. Existing techniques assume inter-scenario synchronizations between two scenarios are encoded by the initial tokens they have on common channels. For such common channels to exist, the corresponding source and destination actors must also exist in both scenarios. This is obviously a restrictive assumption. Chapter 5 has lifted these restrictive assump-

tions and introduced a generalized throughput analysis approach that allows self-timed unbounded scenarios and inter-scenario synchronizations between arbitrary initial tokens. The chapter has presented different techniques, which have varying trade-offs between accuracy and scalability, to bound the worst-throughput of a set of SDF scenarios.

The end-to-end latency of streaming applications is also more than just a performance metric that need to be optimized on a best-effort basis. It is a real-time requirement that must be met. A tight latency bound needs to be derived, for a given set of resource allocations and scheduler configurations. Deriving such a bound in the presence of dynamism adds another dimension to the latency analysis problem. The start and end points of the latency analysis may belong to two different scenarios. In case of dynamically switching scenarios, the analysis requires to verify if these two points are causally related and if the maximum latency between them is bounded. Chapter 7 tackles this analysis challenge. It formalizes the latency definition in the presence of dynamically switching scenarios. It introduces a sufficient condition on the FSM to guarantee a bounded maximum latency. Then, it presents two techniques to derive a bound to the maximum latency under a periodic source.

The above discussed contributions form an analysis framework that takes a high-level DDN specification of a streaming application as an input. Then, it 1) automatically constructs a FSM-based SADF dataflow model, 2) verifies basic properties such as deadlock-freedom and boundedness, and 3) derives real-time temporal bounds such as worst-case throughput and end-to-end latency, while considering resource sharing in a heterogeneous MPSoC platform. The framework is crafted in such a way that actor code can be automatically generated from the DDN specification so that consistency can be maintained automatically between the implementation and the analysis model. The techniques are implemented in open-source dataflow tools, SDF3 [4] and Caltoopia [2]. The contributions advance the state-of-the-art dataflow-based design methodologies in terms of accuracy, scalability, model expressiveness as well as ease of use.

Yet, there are still some open research questions that would strengthen these contributions and build paths to practical adoption of the framework for industrial use. Some of the main research directions in the context of this thesis are listed as follows for future work.

- **Quasi-Static Order Scheduling**
  The analysis of application mappings, presented in Chapter 6, assumes a static-order (SO) schedule is given for each scenario separately. Often, scenario executions are pipelined and scenario switchings occur only at scenario detection points. This gives rise to a quasi-static-order (QSO) schedule of

the FSM-SADF. Such a schedule can be conveniently constructed from the DDN program, since scenario detection points are explicitly given by the firings of detector actors. The QSO schedule should then be the one to be used for implementation, while the SO schedules of individual scenarios are constructed from the QSO schedule for the mapping analysis. The construction of the QSO schedule, in turn, depends on the allocated buffer-sizes of channels, which brings us to the next future work.

- **Buffer-size Analysis**
  Buffer-size allocation is strongly tied with the timing behavior of the application. The static-order scheduling is also influenced by the allocated buffer-sizes. Works exist that study buffer-throughput trade-offs [87,97,106] of SDF graphs. Similar studies for the FSM-based SADF MoC, however, are not available up until now. A starting point would be to investigate the trade-offs between buffer-size allocation and the worst-case throughput, as well as the maximum end-to-end latency. Moreover, for a given set of buffer-size allocations, the total buffer-size can be minimized through buffer sharing. The analysis techniques, presented in this thesis, are well suited for this. This is because a buffer space that is shared between multiple channels can be easily enforced in the analysis by putting a token with the same identifier on all of these channels.

- **Busy-times across Iterations**
  Improvements in WCRTs in Chapter 6 are achieved by identifying busy-times that are within a single iteration. Tighter temporal bounds can be further obtained if the busy-time analysis is extended across iterations. This is possible, for instance, if we take a look at the end-to-end latency analysis in Chapter 7. The characterization matrix of a state-sequence is computed after analyzing each scenario of the sequence separately. If, instead, the matrix is constructed directly from a symbolic simulation of a single execution of the sequence, a matrix that gives tighter bounds can be obtained. This is because the symbolic execution can make use of busy times that extend across multiple iterations of different scenarios.

- **Code Generation**
  The framework discussed in this thesis covers temporal analysis, based on FSM-SADF models that are constructed from a DDN specification. The DDN specification is also intended as a programming model, from which implementation code can be generated. During the course of this work, we have experimented with the Caltoopia [2] compiler to generate C code directly from a DDN specification, written in CAL actor language. CAL

aims at platform-independent application programming by separating the compilation process into two phases: *frontend* and *backend*. The frontend parses the input CAL program and generates an intermediate representation (IR), which is a common input to different types of backend compilations. The backend compilation is platform-dependent and generates code to a specific target. Different backend compilation paths exist today, such as CAL2C for C and CAL2HDL for VHDL/Verilog code generations. This is an interesting infrastructure to efficiently program heterogeneous MPSoCs, which comprise different types of cores, from a target-agnostic application specification. The temporal analysis presented in this thesis comes into play here, guiding the designer and verifying basic and temporal properties.

Bibliography

[1] Bound-T, Time and Stack analyser. http://www.bound-t.com/.

[2] Caltoopia, CAL Compiler. https://github.com/Caltoopia.

[3] Open RVC-CAL Compiler. http://orcc.sourceforge.net/.

[4] SDF3 - Synchronous Dataflow for Free. http://www.es.ele.tue.nl/sdf3/.

[5] 3GPP TS 36.211 V8.6.0: Physical Channels and Modulation, 2009.

[6] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 3–14, Aug 2008.

[7] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi. Building Timing Predictable Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, Mar. 2014.

[8] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1993.

[9] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable embedded multiprocessor system design. In H. Schepers, editor, *Software and Compilers for Embedded Systems*, volume 3199 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2004.

[10] K. van Berkel. Multi-core for Mobile Phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[11] K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Appl. Signal Process.*, 2005:2613–2625, Jan. 2005.

[12] P. van Stralen and A. Pimentel. Scenario-based design space exploration of mpsocs. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 305–312, Oct 2010.

[13] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, June 2009.

[14] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. *J. Signal Process. Syst.*, 63(2):251–263, May 2011.

[15] J. Boutellier, O. Silven, and M. Raulet. Scheduling of CAL actor networks based on dynamic code analysis. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1609–1612, May 2011.

[16] J. Buck and E. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432 vol.1, April 1993.

[17] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 190–195, 2003.

[18] B. Clerckx, A. Lozano, S. Sesia, C. van Rensburg, and C. Papadias. 3GPP LTE and LTE-Advanced. *EURASIP Journal on Wireless Communications and Networking*, 2009(1):472124, 2009.

[19] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband for

MIMO 4G SDR. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 278–279, Feb 2010.

[20] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 775–780. IEEE, 2012.

[21] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, Oct. 2004.

[22] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):889–899, Oct 1998.

[23] J. Eker and J. Janneck. CAL Language Report: Specification of the CAL actor language. Technical report, University of California, Berkeley, 2003.

[24] emarketer reporter. Worldwide Mobile Phone Users: H1 2014 Forecast and Comparative Estimates. April 2014. http://www.emarketer.com.

[25] S. Fernando, F. Siyoum, Y. He, A. Kumar, and H. Corporaal. MAMPSx: A design framework for rapid synthesis of predictable heterogeneous MPSoCs. In *Rapid System Prototyping (RSP), 2013 International Symposium on*, pages 136–142, Oct 2013.

[26] S. Gaubert. Performance Evaluation of (max,+) Automata. *Automatic Control, IEEE Transactions on*, 40(12):2014–2025, Dec 1995.

[27] M. Geilen. Reduction Techniques for Synchronous Dataflow Graphs. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 911–916, New York, NY, USA, 2009. ACM.

[28] M. Geilen. Synchronous Dataflow Scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, Jan. 2011.

[29] M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, and S. Stuijk. Performance Analysis of Weakly-Consistent Scenario-Aware Dataflow Graphs. Technical report, Eindhoven University of Technology, Eindhoven, The Netherlands, December, 2011. Technical Report ESR-2011-03.

[30] M. Geilen and S. Stuijk. Worst-case Performance Analysis of Synchronous Dataflow Scenarios. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 125–134, New York, NY, USA, 2010. ACM.

[31] M. Geilen, S. Stuijk, and T. Basten. Predictable Dynamic Embedded Data Processing. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 320–327, July 2012.

[32] S. J. Geuns, J. P. Hausmans, and M. J. Bekooij. Automatic Dataflow Model Extraction from Modal Real-time Stream Processing Applications. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 143–152, New York, NY, USA, 2013. ACM.

[33] S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij. Sequential Specification of Time-aware Stream Processing Applications. *ACM Trans. Embed. Comput. Syst.*, 12(1s):35:1–35:19, Mar. 2013.

[34] A. Ghamarian, M. C. W. Geilen, T. Basten, B. Theelen, M. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, pages 68–75, Nov 2006.

[35] A. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36, June 2006.

[36] A. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196, Aug 2007.

[37] A. Girault *et. al.* Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 1999.

[38] K. Goossens, J. Dielissen, and A. Radulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *Design Test of Computers, IEEE*, 22(5):414–421, Sept 2005.

[39] R. de Groote, J. Kuper, H. Broersma, and G. J. M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 29–38, Sept 2012.

[40] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 565–568, April 2009.

[41] O. Gustafsson, K. Amiri, D. Andersson, A. Blad, and C. Bonnet *et. al.* Architectures for cognitive radio testbeds and demonstrators - An overview. In *Cognitive Radio Oriented Wireless Networks Communications (CROWNCOM), 2010 Proceedings of the Fifth International Conference on*, pages 1–6, June 2010.

[42] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Applying Dataflow Analysis to Dimension Buffers for Guaranteed Performance in Networks on Chip. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 211–212, Washington, DC, USA, 2008. IEEE Computer Society.

[43] B. Heidergott, G. J. Olsder, and J. van der Woude. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications.* Princeton University Press, 2006.

[44] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques, IEEE Proceedings* -, 152(2):148–166, Mar 2005.

[45] H. Holma and A. Toskala. *LTE for UMTS OFDMA and SC-FDMA Based Radio Access.* John Wiley & Sons, Ltd, 2009.

[46] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287–292, Oct 2008.

[47] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):114–129, Mar 2005.

[48] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[49] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, CODES '00, pages 13–17, New York, NY, USA, 2000. ACM.

[50] J. Kim, H. Oh, J. Choi, H. Ha, and S. Ha. A novel analytical method for worst case response time estimation of distributed embedded systems. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, May 2013.

[51] P. Kourzanov, O. Moreira, and H. J. Sips. Disciplined Multi-core Programming in C. In *PDPTA*, pages 346–354. CSREA Press, 2010.

[52] E. Lee and D. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987.

[53] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[54] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[55] A. Lele, O. Moreira, and P. J. Cuijpers. A New Data Flow Analysis Model for TDM. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 237–246, New York, NY, USA, 2012. ACM.

[56] I. S. Liu. *Precision Timed Machines*. PhD thesis, University of California, Berkeley, 2012.

[57] D. Martin-Sacristan, J. Monserrat, J. Cabrejas-Penuelas, D. Calabuig, S. Garrigas, and N. Cardona. On the Way towards Fourth-Generation Mobile: 3GPP LTE and LTE-Advanced. *EURASIP Journal on Wireless Communications and Networking*, 2009(1):354089, 2009.

[58] M. Mattavelli, J. Janneck, and M. Raulet. MPEG Reconfigurable Video Coding. In *Handbook of Signal Processing Systems*, pages 43–67. Springer US, 2010.

[59] O. Moreira. *Temporal Analysis and Scheduling of Hard-Real Time Radios on a Multi-processor*. PhD thesis, Eindhoven University of Technology, 2011.

[60] O. Moreira and M. J. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1):083710, 2007.

[61] B. Moyer. *Real World Multicore Embedded Systems.* Elsevier, 2013.

[62] D. Nadezhkin and T. Stefanov. Automatic derivation of polyhedral process networks from while-loop affine programs. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 102–111, Oct 2011.

[63] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of Different System Abstractions on the Performance Analysis of Distributed Real-time Systems. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 193–202, New York, NY, USA, 2007. ACM.

[64] A. Pimentel, L. Hertzbetger, P. Lieverse, P. van der Wolf, and E. Deprettere. Exploring embedded-systems architectures with Artemis. *Computer*, 34(11):57–63, Nov 2001.

[65] A. D. Pimentel, C. Erbas, and S. Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Trans. Comput.*, 55(2):99–112, Feb. 2006.

[66] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for Rapid Prototyping. In *Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, RSP '08, pages 17–23, Washington, DC, USA, 2008. IEEE Computer Society.

[67] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 184–189, 2003.

[68] P. Poplavko. *An accurate analysis for guaranteed performance of multiprocessor streaming applications.* PhD thesis, Technische Universiteit Eindhoven (TU/e), 2008.

[69] P. Poplavko, M. Geilen, and T. Basten. Predicting the Throughput of Multiprocessor Applications under Dynamic Workload. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 282–288, Oct 2010.

[70] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 506–513, 2002.

[71] H. Salunkhe, O. Moreira, and K. van Berkel. Mode-Controlled Dataflow Based Modeling and Analysis of a 4G-LTE Receiver. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 212:1–212:4. European Design and Automation Association, 2014.

[72] S. Schliecker and R. Ernst. A Recursive Approach to End-to-end Path Latency Computation in Heterogeneous Multiprocessor Systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 433–442, New York, NY, USA, 2009. ACM.

[73] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, pages 185–190, New York, NY, USA, 2008. ACM.

[74] S. Schliecker, S. Stein, and R. Ernst. Performance Analysis of Complex Systems by Integration of Dataflow Graphs and Compositional Performance Analysis. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[75] F. Siyoum, B. Akesson, S. Stuijk, K. Goossens, and H. Corporaal. Resource-Efficient Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 309–318, Aug 2011.

[76] F. Siyoum, B. Akesson, S. Stuijk, K. Goossens, and H. Corporaal. Dataflow Model for Credit-Controlled Static-Priority Arbitration. Technical report, Eindhoven University of Technology, Eindhoven, The Netherlands, October 2010. Technical Report ESR-2010-03.

[77] F. Siyoum, M. Geilen, and H. Corporaal. Symbolic Analysis of Dataflow Applications Mapped Onto Shared Heterogeneous Resources. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 127:1–127:6, New York, NY, USA, 2014. ACM.

[78] F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal. Automated Extraction of Scenario Sequences from Disciplined Dataflow Networks. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 47–56, Oct 2013.

[79] F. Siyoum, M. Geilen, O. Moreira, and H. Corporaal. Worst-case Throughput Analysis of Real-time Dynamic Streaming Applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '12, pages 463–472, New York, NY, USA, 2012. ACM.

[80] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. Analyzing Synchronous Dataflow Scenarios for Dynamic Software-defined Radio Applications. In *System on Chip (SoC), 2011 International Symposium on*, pages 14–21, Oct 2011.

[81] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000.

[82] T. Stefanov and E. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-level Design. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 90–96, New York, NY, USA, 2003. ACM.

[83] M. Steine, M. Bekooij, and M. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 37–44, Aug 2009.

[84] D. Stiliadis and A. Varma. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. 6(5), 1998.

[85] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[86] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[87] S. Stuijk, M. Geilen, and T. Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. In *Computers, IEEE Transactions on*, USA, 2008. IEEE.

[88] S. Stuijk, M. Geilen, and T. Basten. A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 548–555, Sept 2010.

[89] B. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, July 2006.

[90] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 127–136, New York, NY, USA, 2009. ACM.

[91] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, Apr. 1994.

[92] K. W. Tindell *et. al.* An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Syst.*, 1994.

[93] W. Tong, O. Moreira, R. Nas, and K. Van Berkel. Hard-real-time scheduling on a weakly programmable multi-core processor with application to multi-standard channel decoding. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 151–160, April 2012.

[94] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: A Tool for Improved Derivation of Process Networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, Jan. 2007.

[95] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Dynamic Dataflow. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 319–326, Jan 1996.

[96] M. Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications.* PhD thesis, 2009.

[97] M. Wiggers, M. Bekooij, and G. J. M. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 640–645, March 2008.

[98] M. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling Run-time Arbitration by Latency-rate Servers in Dataflow Graphs. In *Proceedings of the 10th International Workshop on Software &; Compilers for Embedded Systems*, SCOPES '07, pages 11–22, New York, NY, USA, 2007. ACM.

[99] M. Wipliez. *Compilation infrastructure for dataflow programs*. PhD thesis, INSA of Rennes / IETR (France), 2010.

[100] M. Wipliez and M. Raulet. Classification and transformation of dynamic dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 303–310, Oct 2010.

[101] Y. Yang. *Exploring Resource/Performance Trade-offs for Streaming Applications on Embedded Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2012.

[102] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Iteration-based trade-off analysis of resource-aware sdf. In *Proceedings of the 2011 14th Euromicro Conference on Digital System Design*, DSD '11, pages 567–574, Washington, DC, USA, 2011. IEEE Computer Society.

[103] C. Zebelein, J. Falk, C. Haubelt, and J. Teich. Classification of general data flow actors into known models of computation. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 119–128, June 2008.

[104] J. Zhai, H. Nikolov, and T. Stefanov. Modeling adaptive streaming applications with parameterized polyhedral process networks. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 116–121, June 2011.

[105] J. Zhu, I. Sander, and A. Jantsch. Performance analysis of reconfiguration in adaptive real-time streaming applications. In *Embedded Systems for Real-Time Multimedia, 2008. ESTImedia 2008. IEEE/ACM/IFIP Workshop on*, pages 53–58, Oct 2008.

[106] J. Zhu, I. Sander, and A. Jantsch. Buffer Minimization of Real-time Streaming Applications Scheduling on Hybrid CPU/FPGA Architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1506–1511, 2009.

Curriculum Vitae

Firew Siyoum was born in Addis Ababa, Ethiopia, on June 14, 1982. He completed his high school studies in 2000 from Medhanialem Secondary School in Addis Ababa. He received a Bachelor's degree in Electrical Engineering in 2005 from Mekele University, Ethiopia. Until 2007, he was teaching programming languages and digital electronics courses at MicroLink IT College in Addis Ababa. In 2009, he received a Master's degree in Embedded Systems from Eindhoven University of Technology (TU/e), the Netherlands. The focus of his Master's thesis was transaction-level modeling and simulation of multi-core embedded systems, which he carried out at Recore Systems, the Netherlands. After obtaining his MSc, he joined the Electronics Systems (ES) group at the Electrical Engineering Department of TU/e as a scientific programmer. As of September 2010, he started working towards a Ph.D. degree in the ES group. In the fall of 2012, he did a four-months internship at Ericsson Research in Lund, Sweden, as part of HiPEAC industrial PhD internships program. The central theme of his PhD research has been the design-time temporal analysis of dynamic embedded streaming applications, which led to the results presented in this thesis.

List of Publications

First Author

- F. Siyoum, M. Geilen, and H. Corporaal. Symbolic Analysis of Dataflow Applications Mapped Onto Shared Heterogeneous Resources. In *Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 127:1-127:6, San Francisco, California, USA, June 2014.

- F. Siyoum, M. Geilen, and H. Corporaal. End-to-end Latency Analysis of Dynamic Dataflow Applications Mapped on Multi-core Targets. *(Under review for journal publication)*, 2014.

- F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal. Automated Extraction of Scenario Sequences from Disciplined Dataflow Networks. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 47-56, Portland, Oregon, Oct 2013.

- F. Siyoum, M. Geilen, O. Moreira, and H. Corporaal. Worst-case Throughput Analysis of Real-time Dynamic Streaming Applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '12, pages 463-472, Tampere, Finland, Oct 2012.

- F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. Analyzing Synchronous Dataflow Scenarios for Dynamic Software-defined Radio Applications. In *System on Chip (SoC), 2011 International Symposium on*, pages 14-21, Tampere, Finland, Oct 2011.

- F. Siyoum, B. Akesson, S. Stuijk, K. Goossens, and H. Corporaal. Resource-Efficient Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 309-318, Toyama, Japan, Aug 2011.

- F. Siyoum, B. Akesson, S. Stuijk, K. Goossens, and H. Corporaal. Dataflow Model for Credit-Controlled Static-Priority Arbitration. *Technical report, Eindhoven University of Technology, Eindhoven*, Technical Report ESR-2010-03, The Netherlands, October 2010.

Co-author

- Shakith Fernando, Firew Siyoum, Yifan He, Akash Kumar, and Henk Corporaal. MAMPSx: A Design Framework for Rapid Synthesis of Predictable Heterogeneous MPSoCs. In *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2013.

- Shakith Fernando, Mark Wijtvliet, Firew Siyoum, Yifan He, Sander Stuijk, Akash Kumar, and Henk Corporaal. MAMPSx: A demonstration of rapid, predictable hmpsoc synthesis. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on. IEEE*, 2013.

- Roel Jordans, Firew Siyoum, Sander Stuijk, Akash Kumar, and Henk Corporaal. An Automated Flow to Map Throughput Constrained Applications to a MPSoC. In *In Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, 2011.

Acknowledgments

Pursuing a PhD is a lengthy and demanding undertaking, which is filled with many ups and downs. As there are times of delight and success, there are also moments of challenge and frustration. I am grateful to the support of several people, who helped me in different ways to get through this adventuresome journey.

First and foremost, I would like to thank my promotor Henk Corporaal for all of his guidance, support and encouragement over the years. After completing my master's thesis under his supervision, Henk brought me to the Electronic Systems (ES) group to work as a scientific programmer on a temporary basis. It was a great opportunity that introduced me to the world of dataflow modeling and further inspired me to join the COBRA project as a PhD student to carry out research in the area. I am grateful to Henk for the useful discussions in the past years, which have always been valuable inputs to my research and were key aids to guide my research in the right direction. I have also truly enjoyed how Henk elegantly balances quality work and research freedom with a friendly approach towards students. Likewise, I am also deeply grateful to my copromotor Marc Geilen, who has been my mentor in the past four years. His sincere feedback, insightful comments and keen eye for details have been indispensable inputs to the final quality of my works. Marc is quite friendly and helpful, who always made time for my meeting requests (even through Skype calls, when I was away for months). It was a great pleasure working under his supervision.

I would like to extend my uttermost gratitude to the members of my PhD defence committee: Andy Pimentel, Axel Jantsch, Kees van Berkel and Marco Bekooij, for spending their valuable time to review the draft of this dissertation as well as for their constructive feedback and comments.

During my first year at the ES group, I had the privilege to work with Sander Stuijk, Akash Kumar and Roel Jordan in the PreMaDoNa project. I am grateful to Sander for all of his assistance in the past years. Specially, I want to thank him for pointing me to the pessimism problem of the $\mathcal{LR}$ dataflow model of CCSP, which eventually led to a publication we wrote together. I am also indebted to Benny Akesson in this work for thoroughly reviewing the manuscript and for the valuable motivation. I acknowledge Benny and Kees Goossens in this work for the different academic writing techniques I learned from them.

It was also a pleasure to collaborate with members of the COBRA project from ST/Ericsson: Kees van Berkel, Orlando Moreira, Wei Tong, Hrishikesh Salunkhe and Alok Lele. I am thankful to Orlando for helping me with the dataflow modeling of LTE and for introducing me with the CAL actor language. The later, in particular, was one of the reasons that motivated me to apply for an internship at Ericsson in the fall of 2012 to work on the Caltoopia CAL compiler. Many thanks also to the Caltoopia team in Lund. I gratefully acknowledge Johan Eker, Carl von Platen and Harald Gustafsson for giving me the internship opportunity as well as for their unrestrained assistance and kind hospitality during my stay in Lund. I am also indebted to Patrik Lantiz, Christoffer Jerkeby, Song Yuan and other team members for making my time in Lund memorable.

My heartfelt gratitude also goes to the chairs of the ES group: Ralph Otten and Twan Basten, and the secretaries of the group: Marja Regels de Mol, Rian van Gaalen and Margot Gordon, who have always been cooperative, attentive and welcoming. Special thanks to Rian for organizing the weekly "uurtje Nederland"s and for the occasional "uitjes", which were always delightful and a lot of fun.

During my stay at the ES group, I was greatly fortunate to know and work with so many earnest colleagues, all of whose names I can not mention here. Thank you all for making my stay in the ES group enjoyable and memorable! It was a pleasure working with you! My special thanks goes to fellow office mates and "poker night" comrades: Davit Mirzoyan, Manil Dev Gomony, Marcel Stein, Karthik Chandrasekar, Cedric Nugteren, Sven Goossens, Roel Jordan and Luc Vosters for creating a pleasant work environment and for organizing entertaining social events and poker nights.

I also want thank my family for their unconditional love and support; specially to my mother Atsede and my sisters Yenework, Selam and Nitsuh. I am also deeply thankful to Selam Demewoz as well as to Azeb and her family, for spicing up my life in the Netherlands. Spending time with you has always been relaxing and fun. Last but not least, my deepest appreciation goes to my loving wife Tiblets Demewez for bringing joy into my life and for standing by my side through all the good and bad times. You have been a compatriot, a good friend, a loving wife and, of course, a great cook, who made my life in a foreign land feel like home.